Delft Students on Software Architecture: DESOSA 2018

Deursen, Arie van; Zaidman, Andy; Aniche, Maurício; Clark, Liam; Weterings, Gijs; Kharisnawan, Romi

Publication date
2018

Citation (APA)
Deursen, A. V., Zaidman, A., Aniche, M., Clark, L., Weterings, G., & Kharisnawan, R. (2018). *Delft Students on Software Architecture: DESOSA 2018*. (DESOSA; Vol. 4). Delft University of Technology. https://delftswa.gitbooks.io/desosa2018/

Delft Students on Software Architecture

DESOSA 2018

Table of Contents

Delft Students on Software Architecture: DESOSA 2018

Arie van Deursen, **Andy Zaidman**, **Maurício Aniche**, **Liam Clark**, **Gijs Weterings** and **Romi Kharisnawan**.
Delft University of Technology, The Netherlands, July, 2018

We are proud to present the fourth edition of *Delft Students on Software Architecture*, a collection of 20 architectural descriptions of open source software systems written by students from Delft University of Technology during a master-level course that took place in the spring of 2018.

In this course, teams of approximately 4 students could adopt an open source project of choice on GitHub. The projects selected had to be sufficiently complex and actively maintained (one or more pull requests merged per day).

During an 8-week period, the students spent one third of their time on this course, and engaged with these systems in order to understand and describe their software architecture.

Inspired by Amy Brown and Greg Wilson's Architecture of Open Source Applications, we decided to organize each description as a chapter, resulting in the present online book.

Recurring Themes

The chapters share several common themes, which are based on smaller assignments the students conducted as part of the course. These themes cover different architectural 'theories' as available on the web or in textbooks. The course used Rozanski and Woods' Software Systems Architecture, and therefore several of their architectural viewpoints and perspectives recur.

The first theme is outward looking, focusing on the use of the system. Thus, many of the chapters contain an explicit stakeholder analysis, as well as a description of the context in which the systems operate. These were based on available online documentation, as well as on an analysis of open and recently closed (GitHub) issues for these systems.

A second theme involves the development viewpoint, covering modules, layers, components, and their inter-dependencies. Furthermore, it addresses integration and testing processes used for the system under analysis.

A third recurring theme is technical debt. Large and long existing projects are commonly vulnerable to debt. The students assessed the current debt in the systems and provided proposals on resolving this debt where possible.

Besides these common themes, students were encouraged to include an analysis of additional viewpoints and perspectives, addressing, e.g., security, privacy, regulatory, evolution, or product configuration aspects of the system they studied.

First-Hand Experience

Last but not least, all students made a substantial effort to try to contribute to the actual projects. With these contributions the students had the ability to interact with the community; they often discussed with other developers and architects of the systems. This provided them insights in the architectural trade-offs made in these systems.

Student contributions included documentation changes, bug fixes, refactorings, as well as small new features. A list of contributions accepted by the projects under study is provided in the dedicated contributions chapter.

Feedback

While we worked hard on the chapters to the best of our abilities, there might always be omissions and inaccuracies. We value your feedback on any of the material in the book. For your feedback, you can:

- Open an issue on our GitHub repository for this book.
- Offer an improvement to a chapter by posting a pull request on our GitHub repository.
- Contact @delftswa on Twitter.

- Send an email to Arie.vanDeursen at tudelft.nl.

Acknowledgments

We would like to thank:

- Our 2018, guest speakers, offering students an industrial perspective on software architecture: Bert Wolters, Sander Knape, Allard Buijze, and Bob Bijvoet.
- All open source developers who helpfully responded to the students' questions and contributions.
- The excellent gitbook toolset and gitbook hosting service making it easy to publish a collaborative book like this.

Previous DESOSA editions

1. Arie van Deursen, Maurício Aniche, Andy Zaidman, Valentine Mairet, Sander van den Oever (editors). Delft Students on Software Architecture: DESOSA 2017, 2017.
2. Arie van Deursen, Maurício Aniche, Joop Aué (editors). Delft Students on Software Architecture: DESOSA 2016, 2016.
3. Arie van Deursen and Rogier Slag (editors). Delft Students on Software Architecture: DESOSA 2015. DESOSA 2015, 2015.

Further Reading

1. Arie van Deursen, Maurício Aniche, Joop Aué, Rogier Slag, Michael de Jong, Alex Nederlof, Eric Bouwers. A Collaborative Approach to Teach Software Architecture. 48th ACM Technical Symposium on Computer Science Education (SIGCSE), 2017.
2. Arie van Deursen, Alex Nederlof, and Eric Bouwers. Teaching Software Architecture: with GitHub! avandeursen.com, December 2013.
3. Amy Brown and Greg Wilson (editors). The Architecture of Open Source Applications. Volumes 1-2, 2012.
4. Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

Copyright and License

Akka - Build powerful reactive, concurrent, and distributed applications more easily

By: Thomas Smith, Carsten Griessmann, Martijn Steenbergen, Remi van der Laan

Abstract

Akka is a toolkit for building highly concurrent, distributed applications. It was created by a company now called Lightbend, which is also behind the Scala language, and is maintained by a core team of seven people employed by them as well. After a thorough analysis from various perspectives and views of this software system, we concluded there are few problems, though there are some possibilities for improvement. Notably, there are the problems that come along with binary compatibility, which is mentioned by the core team themselves as the largest time intensive burden for technical debt. Besides analyzing the current architecture, we provide descriptions of its evolution and concurrency in the delivery of messages between actors. This chapter serves as a high-level overview for people interested in Akka and to suggest possible improvements to the project maintainers.

Introduction

Akka is a set of libraries for building concurrent, distributed and message-driven applications for Java and Scala. It allows users to build applications in different programming models, such as stream programming or pub/sub, on the Actor model. The actor model aims to take the core principle of OOP, sending messages to objects, and make it safe in the face of concurrency. Akka focuses on simplicity, resilience and performance which has made it widely adopted by large organizations such as eBay, Twitter and Walmart. It is written in Scala and provides bindings to Java as well. It is one of the biggest open source projects within Scala and it keeps evolving.

In this chapter, we aim to provide insight into the Akka project and provide a high-level understanding of its underlying architecture. We start by giving an overview of its stakeholders and the context surrounding Akka. Then an analysis of its architecture and design is given through the development viewpoint, which additionally describes the code organization and standards. This is followed by looking into the evolution of Akka. We then provide more insights in how Akka solves the problem of building parallel application through the concurrency view. We finish by looking at some technical debt that we found.

Stakeholders

Akka is lead and funded by Lightbend. Lightbend, formerly called TypeSafe [1], is a company founded by Martin Odersky (the creator of the Scala language), Jonas Bonér (creator of the Akka framework) and Paul Phillips (who has left the company [2]). Next to Lightbend Akka also has a lot of other stakeholders, that are interested in the success of the product. In this chapter we will identify those using the categories of the book *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*.

Acquirers

Acquirers are the stakeholders that fund the project. In the case of Akka that is the company Lightbend. It employs the core team that oversees the development and is currently listed as their only sponsor [8].

Assessors

Assessors oversee whether Akka meets all legal standards. The main assessors that we identified are the license holders of the dependencies of Akka, such as Oracle (Java/JVM), Lightbend (Scala/SBT), Google (Protobuf) and Netty. Next to that there is probably a legal department within Lightbend that also has that role for Akka. This unfortunately cannot be verified due to it not being transparent in the open source project.

Communicators

Communicators are the ones that explain and document the system so that it can be used by others. The most important communicator for Akka is Lightbend. They are selling consultancy and training services for it, which is an important part of their business model [3]. Next to that, the extensive documentation of Akka can be contributed to by anyone, but most changes are added by its core team.

Developers

Since Akka is an open source project and has existed for many years [4], there have been many developers who understand the architecture and make it through the whole development cycle to make a contribution. That contribution can be anything like a new feature, a bug-fix or even a refactor. This makes them not only developers but also maintainers and testers, since when making contribution they also have to test it. It is notable that many key contributors are also Lightbend employees and part of the Akka core team. Most of those members made the releases on the Akka GitHub repository.

Production engineers

The production engineer in this case is the one that is providing Akka's development infrastructure such as the build servers and the CI environment. This is again Lightbend. Things such as deployment itself is the responsibility of the users since Akka is just a library.

Suppliers

Akka is a quite low-level library and therefor not reliant on suppliers that could be viewed as stakeholder. Their main suppliers like the JVM or Scala are not reliant on Akka being successful so therefor are omitted here.

Support staff

Lightbend provides paid support for Akka for Lightbend subscription members [5]. Other (visible) support questions are handled inside GitHub issues or online communities like Gitter [7].

Users

The users of Akka can be seen as developers of other projects, such as PayPal, Zalando, Wehkamp and Walmart [6]. These developers directly use of Akka to build business applications and have concerns about its functionality. They also are responsible of the correct deployment of their application.

System Administrators

System Administrators are not directly visible since Akka is not an in-house developed enterprise product. It is built and released as a library, but this does not require a system administrator.

Competitors

Akka's main competitors are other solutions in the space of distributed Actor systems such as Erlang, Akka.net and Akka.js, but also other distributed solutions like Kafka.

Reviewers/Integrators

The reviewers and integrators are responsible for maintaining quality and consistency within the codebase of Akka through reviewing pull-requests. Here those are the members of core team that were active in the past 6 months:

- @patriknw
- @johanandren
- @ktoso
- @raboof
- @2m

Power interest relations

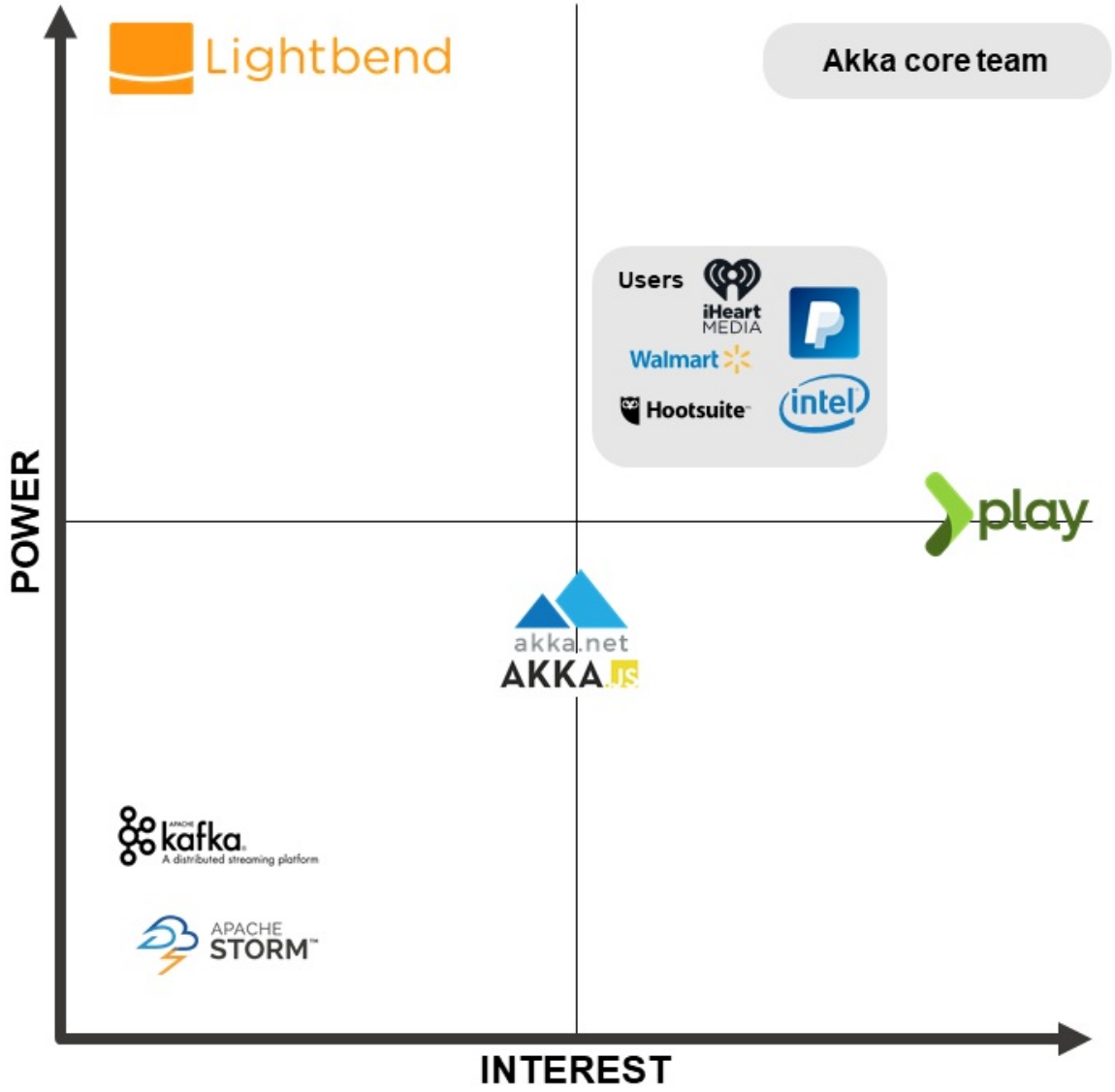


Figure 1 - *Power to interest graph for Akka's stakeholders.*

To visualize the importance of all different stakeholders in comparison with their interest in Akka's development, a power to interest diagram was created. The most important ones here are the core development team, which always has the final say on the project's development, and again Lightbend.

Context View

The Context view focuses on the relationships and dependencies between Akka and external entities.

Context model diagram

To visualize Akka's interactions with its environment, the following diagram was made.

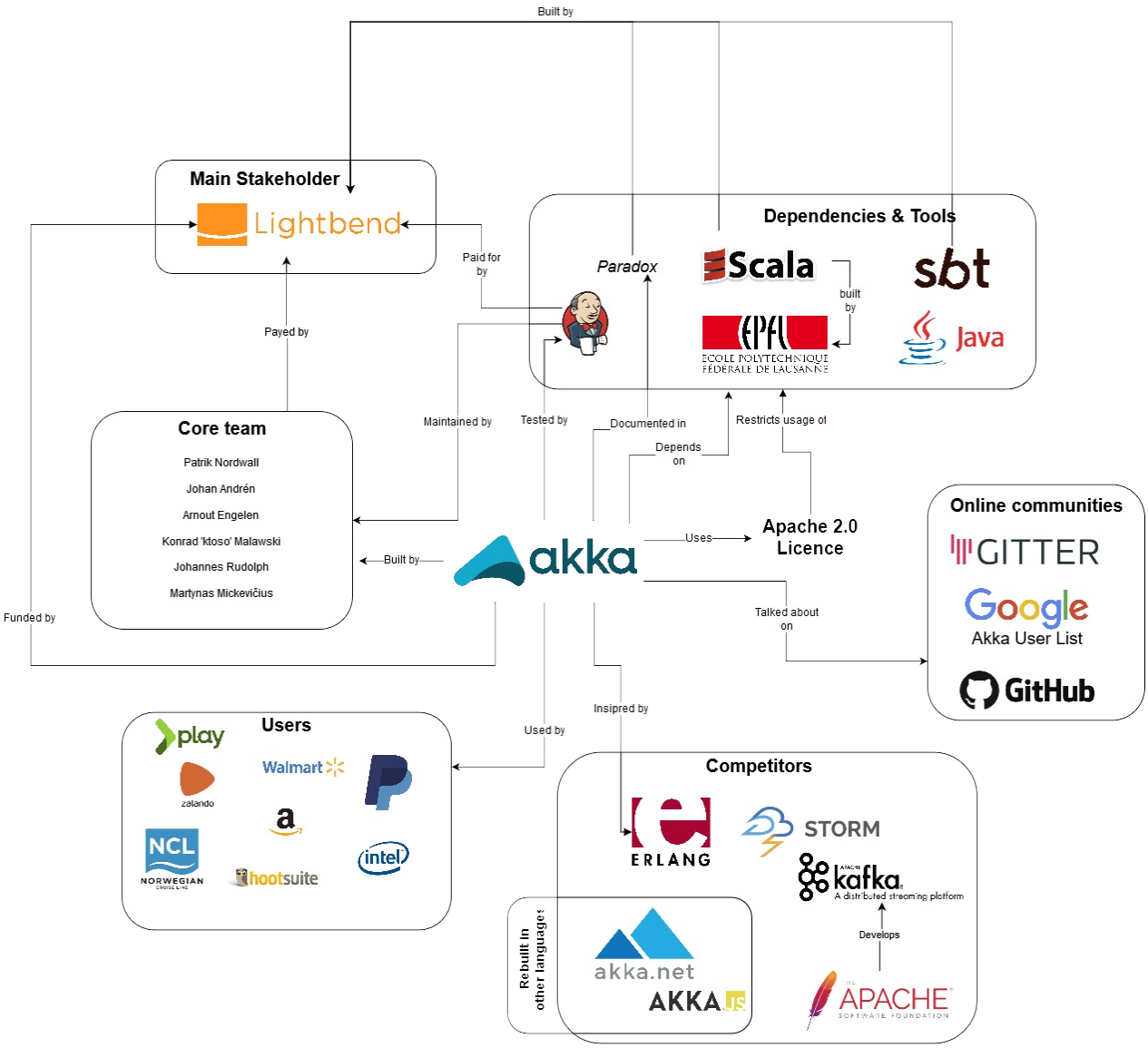


Figure 2 - *The context view for Akka, showing the external parties and their relationship to it.*

From this diagram, one very important item is to be seen and that is the ubiquity of Lightbend in this project. From funding the CI server, developers, documentation tools, but also maintaining the language (Scala) and the build-tool (SBT) this framework is written in. Akka's main dependencies are Scala, SBT and the JVM. Those are baked into the architecture of Akka in such a way that cannot be changed. Akka depends on several of those tools that are fundamental to the Scala ecosystem. Therefore, the general direction and development of Scala, JVM and SBT heavily influence the possibilities for Akka. So Lightbend's influence on Scala potentially reduces the risk of problems occurring in the ecosystem that affect Akka. Next to that Akka has several competing libraries in the space of distributed Actor systems. The most noteworthy here is Erlang. Next to being a competitor it's also a big source of inspiration and since Erlang lives in a separate eco-system its competition gets less significant. There are also several online communities that, as mentioned in the stakeholder analysis, work as Akka's free support and as discussion boards.

Development View

Introduction

Akka is a big project and contains a lot of modules. To understand the design constraints and how the project is developed a Development view is made. Here we look at the organization of the modules, analyze standards and processes and the source-code organization within the project.

Module organization

In this part of the development view we will focus on the organization of the modules that Akka is composed of. To perform our analysis, we have used the built-in dependency analysis tools that JetBrains IntelliJ provides. We will first briefly describe the purpose of each module and then dive deeper in their dependencies.

Core modules

The core of Akka consists of two modules that do not depend on each other or other modules. They implement the basic functionality that Akka provides. As a consequence, most other modules depend on these. The `akka-actor` module, the largest module in the project, is responsible for providing a basic framework to the Actor model. The `akka-protobuf` module provides code used for serializing and de-serializing messages that get sent over the network. As the name suggests, this module relies on Google's Protobuf library. Every other Akka module that needs to send anything over the network uses this module for serialization.

akka-persistence

`akka-persistence` enables Actors that contain internal state to persist that state. By default, it uses the key value store levelDB to do that, but it can also be extended with various community plugins to support other stores, where `akka-persistence-tck` implements a technology compatibility kit to ease the way of implementing those. The `akka-persistence-query` module complements `akka-persistence` with an asynchronous query interface that could be used to separate the query side from the write side in a model.

akka-stream

The Actor model can be seen as a more low-level model for stream processing; actors send and receive messages that have to be dealt with in (near) real-time. Because building robust streaming applications on the Actor model can be cumbersome, Akka provides an abstraction around `akka-actor` that provides reliable streaming capabilities.

akka-remote

The `akka-remote` module is responsible for making the actor system location agnostic and allowing actor applications to be distributed over multiple servers. It was designed for peer-to-peer communication and the API can be used to create and lookup actors remotely.

akka-cluster

Built on top of the core and remoting modules, Akka provides tools for managing and sharding clusters which run Akka applications. The `akka-cluster` modules provide fault-tolerant cluster membership, leadership election and failure detection. This module is complemented by several other modules.

- `akka-cluster-sharding` facilitates sharding of actors across machines;
- `akka-cluster-tools` provides a client for actor systems that are not in a cluster to communicate with actors that do reside in a cluster;
- `akka-cluster-metrics` is an extension that provides system health metrics on actors;

akka-*-typed

These modules are an initiative to develop a new set of APIs for each of the above modules. These are more type safe and can be seen as a layer on top of the other modules. For example, it provides a new way of building actors for which the compiler type-checks messages that actors can send and receive, therefore avoiding unsafe runtime casts.

Dependencies of modules

Using the dependency analysis in IntelliJ, we created a dependency matrix which in turn we used to visualize the module structure.

	akka-actor	akka-actor-typed	akka-protobuf	akka-remote	akka-cluster	akka-cluster-metrics	akka-cluster-sharding	akka-cluster-sharding-typed	akka-cluster-tools	akka-cluster-typed	akka-distributed-data	akka-persistence	akka-persistence-query	akka-persistence-tck	akka-persistence-typed	akka-stream	akka-stream-typed	akka-agent	akka-slf4j	akka-contrib	akka-camel	akka-osgi
akka-actor																						
akka-actor-typed	2265																					
akka-protobuf																						
akka-remote	32852		13175													3311						
akka-cluster	7206		6587	268																		
akka-cluster-metrics	1099		2836		147																	
akka-cluster-sharding	3785		2926		265				47		356	517										
akka-cluster-sharding-typed	175	124	269	104	4		81		10	57												
akka-cluster-tools	4218		3180	39	976																	
akka-cluster-typed	410	1300			212				25		1299											
akka-distributed-data	3827		13715	33	3552																	
akka-persistence	16734		1913																			
akka-persistence-query	630											73				287						
akka-persistence-tck	792											815										
akka-persistence-typed	469	1682										183										
akka-stream	10246		2006																			
akka-stream-typed	65	97														78						
akka-agent	28																					
akka-slf4j	183																					
akka-contrib	2389			2																		
akka-camel	1388																					
akka-osgi	113																					

Figure 3 - *The dependency matrix for all modules in Akka.*

From this matrix one can determine that most dependencies are between modules which belong in the same group (`akka-cluster-*`), if we do not look at the dependencies on Akka core. It immediately becomes clear from the table that Akka does not have a layered architecture. The `akka-actor` module, together with `akka-protobuf` forms a core where nearly all other modules depend on. Cluster management modules form an almost perfect clique, indicating that the individual modules are not independently usable and thus that `akka-cluster-*` modules together form one large subsystem. Another notable observation that we can make from this is that the `persistence` modules have a relatively low coupling. We can also clearly see the modules in white, that will be from here on out be classified as `other` , only depend on `akka-actor` and therefore have little impact on the overall architecture.

To convey the high-level architecture as naturally as possible, we abstracted to groups of modules and showing the dependencies of those groups. We compacted the original dependency matrix to only count dependencies between subsystems (like `cluster`), as many modules have the same prefix and therefore it can be said that they together solve a higher-level problem.

	Core	Remote	Cluster	Persist	Stream	Other
Core						
Remote	93%				7%	
Cluster	98%	1%		1%		
Persistence	99%				1%	
Stream	100%					
Other	100%	0%				

Figure 4 - *The high level dependencies between subsystems in Akka.*

This matrix displays what percentage of external references from one of the rows points to one of the columns. For the remoting subsystem, 93% of its external references use one of the core modules. Only 7% use functionality in streaming modules. Note that this matrix is very sparse; subsystems are tightly coupled to the core but relatively loosely coupled to each other.

Some seemingly strange dependencies can also be seen.

- `akka-remote` **depends on** `akka-stream` : This is caused by the new Artery system, a remoting subsystem that will eventually replace the old API. Artery supports sending messages between actors with TLS streams instead of Netty TCP. This is used for network communication in a distributed actor system.
- `akka-cluster-sharding` **depends on** `akka-persistence` : this turns out to be because user-defined shards can also be stored on disk, which makes them persistent after a complete cluster reboot.
- `akka-persistence` **depends on** `akka-stream` : This can be explained by the fact that certain database operations return a stream for performance and convenience.

An interactive dependency visualization can be accessed here.

Standardization of Design

Since Akka is open source and quite large, the core Akka team has defined a set of rules and guidelines in order to maintain a consistent design and improve maintainability in general. We will investigate the most interesting ones here.

User extensibility

Akka provides various classes or subsystems with interfaces with a default implementation or library. In fact, extensibility is encouraged though Akka Extensions, which is how many features such as Typed Actors and Serialization have been implemented. Customized implementations can be specified in the Akka configuration in order to integrate them in a system. More about user extensibility can be found in the Evolution Perspective

Quick overview of standardization

- Most code style standards are standardized automatically. For Scala, the scalariform code style is applied and for Java they use For the Oracle Java Style Guide. Akka also has a lot of scripts to make standardization easier such as e.g. standardizing line endings, indentation and finding documentation errors [9].
- Contributions that add whole new features should be added to the `akka-contrib` module.
- Each module has to be tested and well documented. Testing is done with `ScalaTest` and `ScalaCheck` [10]. A continuous integration server is used to test every new addition to the codebase. [11].
- Large changes should be documented in the official documentation in the `akka-docs/paradox` module.

Binary compatibility

One major and obvious design principle in Akka that can only be found by looking at documentation is binary compatibility [12]. This will be explained further in the Technical Debt section.

Codeline Organization

The source code structure represents the same structure as the previously described module structure. Each module has its own folder which contains the *src* folder as shown in figure 5. The settings, plugins and dependencies for each module are defined in the `build.sbt` configuration file in the project root. It provides the entry point for SBT and there the separate configuration files for building each module are bundled together. These all can be found inside the *project* folder. The *scripts* folder contains all sorts of scripts, for example for code formatting or pull-request validation.

```
akka
|
└──akka-actor
|   └──src
└──akka-agent
|   └──src
|   ... (all module folders)
└──project
└──scripts
└──build.sbt
```

Figure 5 - *The folder structure of Akka.*

Evolution perspective

This section gives a quick overview of the current and future state of Akka, focuses on the flexibility in regard to change and the techniques that are used to accomplish this.

System evolution

Figure 6 shows the major releases and the most noteworthy features they (experimentally) introduced. Before version 2.4, Akka followed the Java or Scala style of versioning: `epoch.major.minor`. Since 2.4 however they follow an approach closer to semantic versioning: `major.minor.patch` [13].

Figure 6 - *The version history of Akka, showing the features they (experimentally) introduced.*

After the initial core Actor pattern was implemented in Akka, many features have been added to satisfy and solve problems for its users. Clusters provide fault-tolerant scalability, streams provide performance in data transfer and processing and persistence provides robustness by allowing the recovery of actor states after restarting. Alpakka, an independent project that will soon get its own full team, provides easy extensibility for connections to external technologies. Akka HTTP, again a independent project, gives users functionality for HTTP integration needs for building their own applications. A more detailed creation story can be found in their five-year anniversary blog post.

Recent and future development

There are still problems being solved and to be solved in the future of Akka. These are likely not goals that were set at the beginning of Akka, but feature requests that have developed over time.

Akka Typed is a new attempt to bring type safety to Akka. The reason why this was not integrated at the start of the project is due to its heavy inspiration from the Erlang language. It introduces typed versions of actors, clusters, and persistence, so that everything is type safe [14]. This brings more compile time safety, which causes less debugging at runtime leading to more productive development. Two other attempts in the past with the same goal have failed and have become deprecated. Typed Actors was too slow due to reflection and not true to messaging; a core principle of Akka. Typed Channels was too complex since it introduced too many operators and it relied on Scala macros. The new Akka Typed is has none of these downsides while introducing benefits, such as actor behaviors acting as a state machine.

Artery is the modern remoting layer, which utilizes UDP instead of TCP for better performance and includes various other optimizations. Currently it remains an experimental feature, but it is to become stable relatively soon [15]. It will replace the old Akka Remoting implementation and it mostly source compatible with it, meaning it can act as a drop-in replacement in many cases [16].

Multi Data Center (DC) is a new experimental feature that provides global scalability and improved stability, with benefits such as serving requests close to a user's proximity and large scale load balancing. This is accomplished by making one Akka Cluster span over multiple data centers [17].

Dealing with change

Akka provides some measures to extend its public API, but also internally strive to integrate facilities for change.

Public API

Akka makes use of variation points in order to allow for specific localized design solutions through Akka Extensions [18]. Standard extension points for storage back-ends in persistent actors can be found in the community plugins repository [19]. Runtime configurations of actor systems can be defined in a type-safe configuration format developed by Lightbend. This allows for configuration of logging, remoting, serializing and much more [20].

Along every major version increase an extensive migration guide is provided, which includes the most notable changes to the public API and the reasoning behind them. The binary compatibility mentioned in Technical Debt forces changes to remain compatible. This causes old features to become deprecated in favor of entirely new features with the same goal, as is the case with Akka Typed.

Internal change

The core team is actively working on creating extensible interfaces; we found a recent issue where many parameters were being replaced with a settings file in order to provide a higher level API for a certain feature.

One more technique is the exclusion of unnecessary additions that can be implemented by users themselves, which we found out first-hand while looking for an issue for our first contribution. They rather provide generic solutions that require some work from their users.

An observation that we personally made was that there is an inconsistent use of features from the Scala language, such as inheritance, which makes it difficult for newcomers to understand the codebase and has negative consequences for the overall maintainability.

Conclusion

Highly flexible systems can bring significant costs in terms of runtime efficiency and performance. However, it doesn't seem like this is the case for Akka; all configuration is processed at compile-time, which does not affect the runtime performance. The techniques they apply to deal with change ensure that the system can evolve while keeping it well maintainable. The architecture is built for both performance and flexibility, allowing Akka developers and its users to easily change and extend features to their pleasing.

Concurrency View

Being a framework for distributed actor systems, Akka must make use of strong concurrency constructs to keep actor systems safe and performant. In this section we explore the main constructs Akka uses to achieve safe and fast concurrency.

We will first look at the core machinery that makes actors work. With the core infrastructure covered, we can then explore how remote messaging between actors fits in the picture.

Actor component and threading model

To support actor concurrency to the full extent, Akka uses a unified threading model for nearly everything that requires tasks to be offloaded from the main thread. Let's consider a simplified component model to illustrate the most important components of an actor:

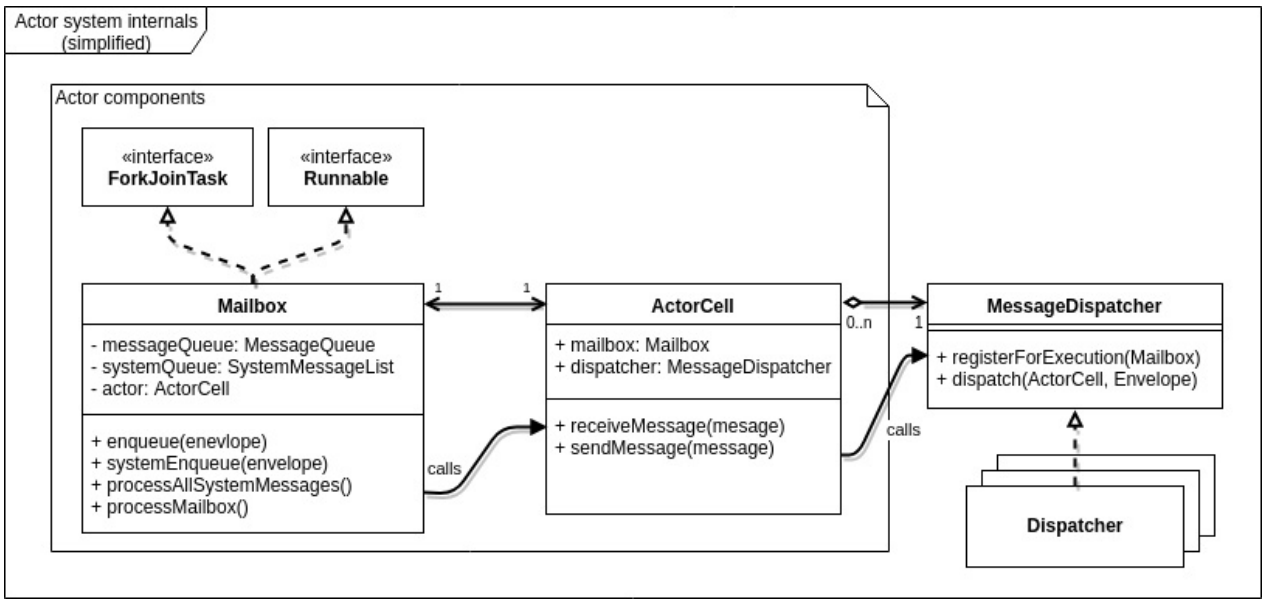


Figure 7 - *A simplified component model.*

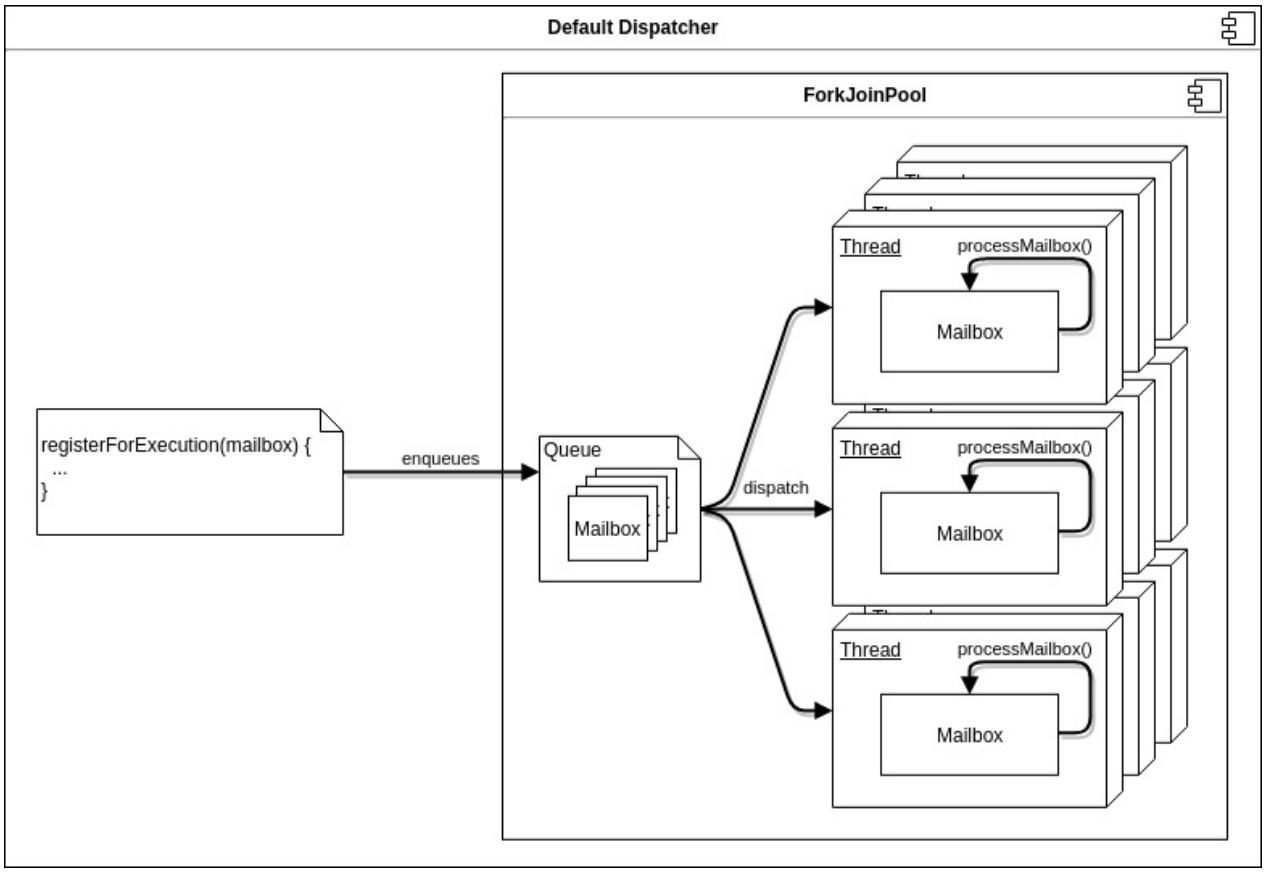


Figure 8 - *The dispatcher threading model.*

Each actor is created with a `Mailbox` and `ActorCell` and is registered with one `MessageDispatcher`. Multiple actors can use different mailbox and dispatcher classes. Note that since `Mailbox` implements `Runnable` and `ForkJoinTask`, it's possible to schedule a mailbox object to run on another thread. Its `run()` method dispatches to the actor's user-defined `receive` method for any message that's processed. In turn, when `sendMessage` gets called on an actor, it will use the registered dispatcher to enqueue the message in the receiver mailbox and schedule the mailbox for execution.

The default dispatcher uses a `ForkJoinPool` is used for scheduling. This is a threading construct that runs tasks on a resizable pool of threads. Because threads get reused as soon as they become free, running an actor is very inexpensive and running many actors automatically utilizes a server's multi-core capabilities.

This model is not only used to run individual actors, but also to schedule any `Future`s that an actor may create.

Scheduler

The `Scheduler` is another concurrency construct that's made available to users by Akka. The scheduler is available in each actor and enables the user to postpone lightweight tasks for a short amount of time. It runs an algorithm based on a revolving wheel of buckets that advances at a fixed tick rate and dispatches tasks it finds in the current bucket to their respective ExecutionContexts. Since it runs tasks in their own ExecutionContext, the tasks themselves run on the execution model that's also used to run the actors. The Scheduler only creates its own thread for running the timer used by the algorithm.

Remote messages

Sending messages between remote actors utilizes the above model. When remoting is enabled, special actors get started that receive messages that cannot be dispatched locally. These endpoint actors are responsible for maintaining the necessary remote connections and serialization. When a remote message is received, the endpoint actor decides which local actor to relay the message to. Multiple endpoints exist for communication over TCP, UDP and TLS.

Synchronization of shared state

In a few key places, state has to be shared between threads for example to communicate the state of a mailbox. Akka achieves this by relying mostly on volatile variables instead of locks to avoid lock contention. An example where this is needed is for actor supervision. Actors are usually organized in hierarchies where one actor is supervised by another. When an actor terminates, it has to terminate all its children and suspend the mailbox. Likewise, when an actor crashes, its parent has to decide whether to stop, resume or recreate the crashed actor and its children. The state to track which children have terminated and whether the mailbox is open or closed (in case of termination) is maintained in volatile variables that are set by means of compare-and-swap operations.

User extensibility

As discussed before, Akka values user-extensibility. This also resonates in their concurrency model. Throughout the entire framework, one framework for execution is used of which many parts are extensible or at least configurable. Mailboxes, dispatchers and thread pools are all extensible to support different application workloads. For example, the default dispatcher can be configured to use any other `ExecutorService` instead of `ForkJoinPool`. One could use `AffinityPool` instead to ensure that each actor always runs on the same thread, or create a custom `ExecutorService`, which may be faster in certain scenarios. The same holds for mailboxes; the message queue used internally can be changed to anything that fits a user's needs. This enables programmers to make their own choices when it comes to concurrency and multi-core performance when building an actor-system.

Technical Debt

Technical debt can be described as the build-up of problems caused by making changes work with easy solutions instead of making them better through more time-consuming approaches. Identifying it is challenging and for this chapter several methods were run. We looked at code hotspots (files that are often changed), test coverage and compiler warnings but were unable to find any noteworthy technical debt. Therefor we decided to focus on the two most interesting subjects that do attribute to technical debt in this section. To also get a better impression of how the maintainers of Akka experience technical debt, we decided to contact them and ask them how they feel about debt in the project.

Binary Compatibility

Akka maintains backwards binary compatibility for minor and patch versions. This means that for users can safely upgrade Akka to a newer minor or patch versions without risking breakage in their products; new JARs are drop-in replacements for older ones.

A consequence of this is that the Akka core team has to carefully manage which parts of their API fall under the compatibility guarantee and which parts do not. The team maintains several ways to signal that an API is not subject to binary compatibility:

- **May change** is used in module descriptions and docs,
- Classes annotated with `/** INTERNAL API */` or with `@InternalAPI`, `@ApiMayChange`, or `DoNotInherit`.

In general, developers of open source projects have to strike a balance between public and internal APIs. Internal APIs can be changed more frequently as there is no risk of introducing breaking changes for the user. However, keeping lot of the APIs internal restricts the users possibilities to extend the system. This may lead to a lot of patch requests for supporting a new use case that wasn't thought of by the developers.

On the other hand, if developers make too much of the API public, they risk severe inertia as users will inevitably depend on all public APIs, making changes to the core hard because some form of backwards compatibility has to be provided.

For Akka specifically, any mistake or change in public API results in that API being deprecated and replaced by something else. However, the deprecated code will have to remain in the project until the next major release, which may take several years. The cost in changing APIs is therefore so high that maintainers are more willing to sacrifice cleanness to avoid this cost. One maintainer mentioned a typical example of this, 'dead code' that is kept in the system to not break the public API: `ClusterSingleton.scala`.

Besides releasing a new major version of Akka more often which may be undesirable, there is no clear and set solution to this problem. Ultimately it comes down to two aspects:

- balance public and internal APIs,
- design new public APIs to be as future-proof as possible.

When asked whether the team can quantify the cost of binary compatibility, the consensus was that this is hard to measure in practice [21]. The Akka team could investigate in methods to make the time they spend on these issues more quantifiable. If the cost of working around public API changes is better known the team may be in a better position to make sound decisions on whether to introduce an API change or sacrifice code quality.

Temporal Coupling

We have used CodeScene to analyze temporal coupling. This refers an analysis of which files are frequently changed together. Temporal coupling can be an indicator for technical debt as it can expose several code smells such as Shotgun Surgery, Code Duplication and Inappropriate Intimacy. After a first run of the analysis we noticed that the results needed some filtering to prevent false positives:

- `*Test.java` and `*Spec.scala` files tend to be tightly coupled with the classes under test so we filtered out any coupling between files where one of the two was a test file.
- Akka features a dsl-like API for both Scala and Java users. Generally this means that when one of the APIs changes, so does to other. We therefore filtered pairs of files with the same name where one in in the `javadsl` folder and the other in the `scaladsl` folder.

The filtered result is shown in figure 9. Lines between files indicate that they are often changed together; thicker lines indicate stronger coupling.

Figure 9 - *A visualization of the temporal coupling in Akka.*

The abstract pattern that we see here is one that applies to many things in the domain of data processing. For many streaming classes, one is a dual to the other: sources create data, sinks reduce data. The way in which they do this may differ, for example, IOSource has a very different implementation compared to AeronSource. But these two classes will generally form a pair where the behavior of one must match the behavior of the other. Another example of where this applies is with serialization. A Serializer class will always be tightly coupled with a Deserializer because the two must work on the same serialization format, causing duplication of code and logic. The cost induced by this coupling can be tough to deal with. A mistake in one of the dual classes can manifest in the other in very unpredictable ways.

Given that there is no easy way around this coupling, the best the Akka team can do is write good tests for these cases and extract as much common code as possible (although the latter will not always be possible). Luckily, the classes in question are well covered by tests.

Conclusion

With over 9 years of development, Akka has grown in many aspects; not only in size, but also in the creation of a solid architecture. It remains flexible and constantly improves its performance and reliability despite of the complex goals it tries to achieve.

We have analyzed Akka from various perspectives and viewpoints to give an insight into the inner workings. It was impressive to see how little actual issues we could find in a project of this size. The architecture is well thought of and the core team behind it are always making proper decisions even if that sometimes leads to more pain in maintaining the project. The success and usage of Akka really prove that. We think it will remain a popular choice for creating distributed applications and are curious on how the project will keep evolving.

References

1. Mark Brewer. Typesafe Changes Name to Lightbend. https://www.lightbend.com/blog/typesafe-changes-name-to-lightbend
2. Paul Phillips. Pacific Northwest Scala 2013 We're Doing It All Wrong. https://www.youtube.com/watch?v=TS1lpKBMkgg
3. Lightbend Consulting Services. https://www.lightbend.com/services/consulting
4. Jonas Bonér. Akka 5 Year Anniversary. https://www.lightbend.com/akka-five-year-anniversary
5. Lightbend Subscription. https://www.lightbend.com/subscription
6. Lightbend Case Studies. https://www.lightbend.com/case-studies
7. The akka/akka Gitter Channel. https://gitter.im/akka/akka
8. Akka. Sponsors. https://doc.akka.io/docs/akka/current/project/links.html#sponsors
9. Akka. Contributing Guide https://github.com/akka/akka/blob/master/CONTRIBUTING.md
10. Akka. Testing Guidelines. http://downloads.lightbend.com/paradox/akka-docs-new/20170511-sidenotes/java/dev/developer-guidelines.html#testing
11. Akka. Continuous Integration https://github.com/akka/akka/blob/master/CONTRIBUTING.md#continuous-integration
12. Akka. Binary Compatibility Rules. https://doc.akka.io/docs/akka/current/common/binary-compatibility-rules.html
13. Akka. Versioning Scheme. https://doc.akka.io/docs/akka/current/common/binary-compatibility-rules.html#change-in-versioning-scheme-stronger-compatibility-since-2-4
14. Konrad Malawski. Networks and Types -- the Future of Akka. https://www.slideshare.net/ktoso/reactive-systems-tokyo-networks-and-types-the-future-of-akka
15. Konrad Malawski. State of Akka @ 2017 - The best is yet to come. https://www.slideshare.net/ktoso/state-of-akka-2017-the-best-is-yet-to-come
16. Akka. Remoting (codename Artery). https://doc.akka.io/docs/akka/2.5/remoting-artery.html
17. Akka. Multi-DC. https://akka.io/blog/2018/01/17/multidc
18. Akka. Akka Extensions. https://doc.akka.io/docs/akka/current/extending-akka.html
19. Akka. Persistence. https://doc.akka.io/docs/akka/current/persistence.html?language=scala
20. Akka. Configuration. https://doc.akka.io/docs/akka/current/general/configuration.html
21. Gitter conversation on Technical Debt. https://gitter.im/akka/dev?at=5aaa41b7bb1018b37ae7ee04

Angular

Blazej Kula	Arvind Chembarpu	Algirdas Jokūbauskas
		

Delft University of Technology

Abstract

Angular is a Free and Open Source Typescript-based framework for developing web applications. It is a complete rewrite from its predecessor AngularJS and is maintained by Google, along with a community of individuals and corporations. It helps users create fast, multi-platform applications in Typescript or Javascript with ease. In this chapter, we will discuss the architecture of the Angular project in detail and try to explain how it is developed. We will be building this analysis on top of the concepts learned in "Software Systems Architecture" by Nick Rozanski and Eoin Woods.

Table of Contents

Introduction

AngularJS was created by Google and released in 2010. The main idea was to decouple HTML DOM manipulation from application logic. It does so by introducing Model-View-Controller architecture to Client Side Javascript code. However, a few years later, in 2014, Google announced Angular 2 - a complete rewrite of AngularJS. It introduced the possibility to code in Typescript (a superset of Javascript) and changed the internal architecture of both itself and applications created with it. It also improved performance and complexity, while also introducing support for building cross platform native mobile apps. However, the lack of backwards compatibility with AngularJS created a lot of controversy among developers. It was fully released in 2016 and has been through many changes since, including a new preview release version 6.0.

We will start our analysis by defining the different stakeholders in the project. We will then provide a contextual view of the application showing its relation to the environment. Furthermore, we will present a development view showing different modules of the application and and design models used. Lastly, we will analyze various kinds of technical debt that have accrued over its lifespan.

Stakeholders

Angular was created and is maintained by Google, along with a large community of individuals and corporations. We compiled this list of stakeholders by analyzing the Angular repository, official website, and Github's Insights & Analytics. In order to provide context and reasoning, we have also provided notes alongside our findings. We will also be using bold text to highlight the actual stakeholders within the analysis.

Primary Stakeholders

In this section, we detail the stakeholders defined by Rozanski and Woods in the course book Software Systems and Architecture.

Acquirers

Google is arguably the most important primary stakeholder, considering how Angular was created by Google engineers for Google's projects. They use Angular internally and externally for a variety of projects, and so OSS contributions can be considered to be continuously "acquired" by Google.

Assessors

Legal compliance is managed by Max Sills (Angular's Open Source Lawyer, employed by **Google**), who can be considered as an assessor. Angular currently uses the MIT license, but used to be distributed under the Apache 2 license - this was changed due to community feedback, as a community-friendly license contributes to the project's success and uptake.

Communicators

Contributors and **Committers** are communicators by default, as every contribution is required to have accompanying documentation and justification. **Google** is also an important communicator, as they have dedicated employees who work on improving documentation and also offer services through the Google Developer Expert program. They also conduct events to promote the project.

Developers

Contributors on GitHub, which includes **Google** and non-Google developers alike, are the development stakeholders. All contributions are made publicly on GitHub and stakeholders are expected to participate to make their voice and needs heard.

Maintainers

Angular is a framework and not a deployable product on its own, so we have considered maintenance of the framework itself in this section. Under this definition, the same contributors who are developers can be considered maintainers, however, not all of them are involved in regular upkeep. **Google** and other **companies** that utilize Angular have a greater investment in keeping Angular running well.

However, it is the **Angular core team** employed by Google, which decides the primary release schedule and milestones.

Suppliers

The **Git SCM** is used to track, version, and control the source code and changes. On top of this, **GitHub** provides the primary platform for code storage, tracking issues, accepting contributions, and making releases. **Google** can also be considered a supplier as they employ the core team, provide necessary infrastructure, and various other resources to the project. **NPM** provides the primary distribution medium for Angular and its dependencies. Angular is developed using **TypeScript**, a superset of plain Javascript.

Support Staff

Users can open Issues on Github to obtain support. **Gitter** provides a chatroom for users to discuss issues and obtain help from their peers. Further, **Google** offers paid professional support to users through their Google Developers Experts program. Google also offers an official Google Group for a mailing-list-like forum. There are multiple third-party communities, as well - on Reddit and on StackOverflow.

System administrators

Angular only provides a framework, so any system infrastructure is independent of the project. The end-users of Angular, who administrate their own Angular-based product can be considered system administrators, but not necessarily in the context of Angular itself.

Testers

All **contributors** are expected to test and ensure that their changes work locally. Further, they must provide useful and effective tests for every change that they propose. **Google** both develops and uses Angular in-house so they also run independent tests to ensure compatibility with their systems, via the Google3 bot (status reported via NgBot). **Users** of the framework also technically test the framework in-use. They are expected to report bugs and provide logs, in order to obtain support.

Users

Web/Application **developers** are the primary users of Angular. **Content-producers** who produce blogs, videos, tutorials, guides, etc. related to Angular can also be considered user-stakeholders. There are also a multitude of **OSS projects** which utilize Angular.

Secondary Stakeholders

In this section, we identify self-defined categories of stakeholders that we consider useful.

Evangelists

Like any other popular OSS project, Angular has its fair share of enthusiasts who introduce and help others in using the framework. They may do this through blog posts, multimedia content, technical guides, textbooks, conferences, and meetups.

Competitors

Competing frontend JavaScript frameworks can also be considered to have a stake in Angular, as its popularity can set the trend for various programming paradigms. Further, they can look to the Angular project for inspiration in planning features, quickly making similar bug or security fixes, or even rethink parts of their own approaches. For example, React, Vue.js, and Ember.

Dependencies

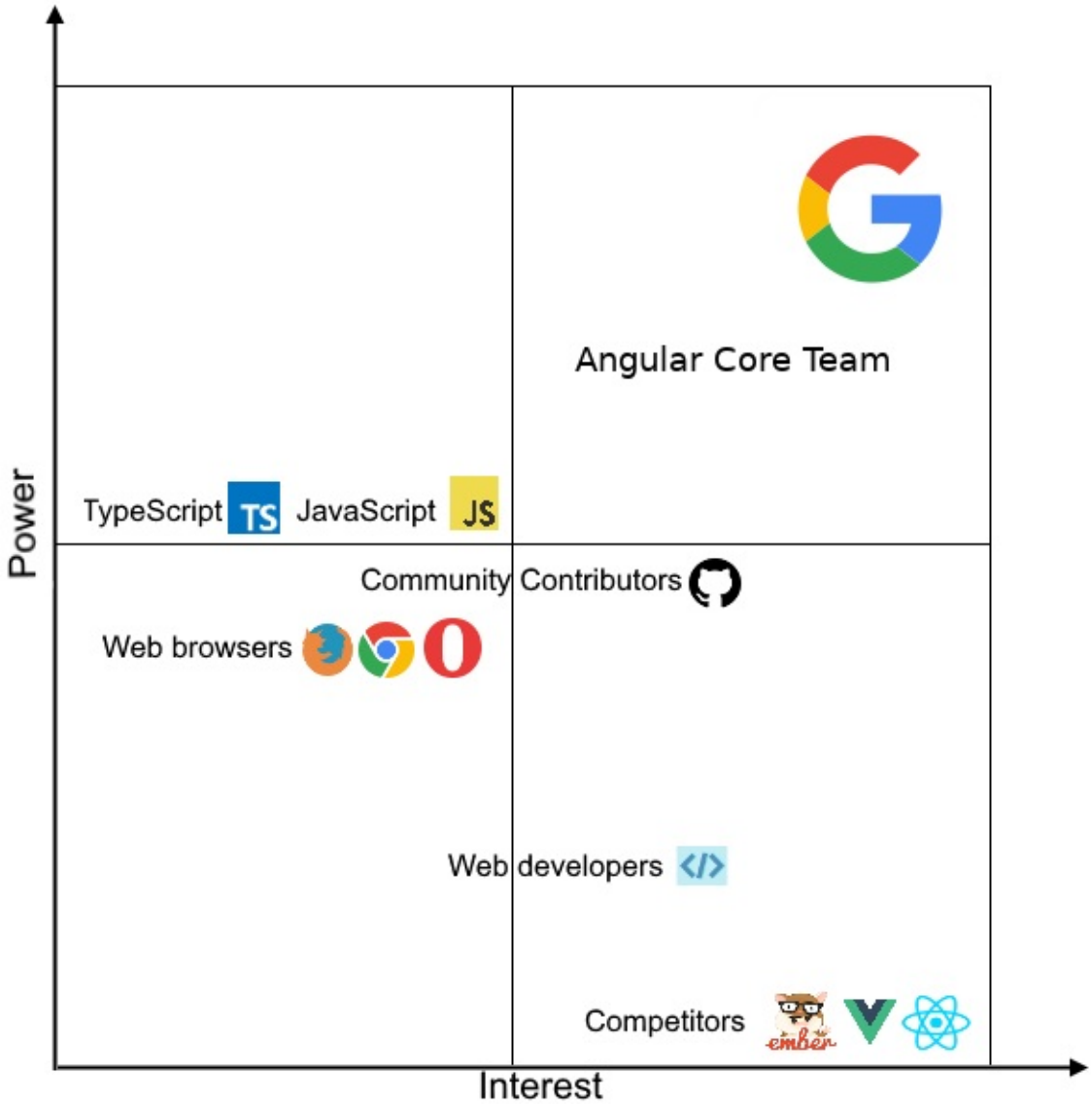
Angular has a number of JS Dependencies which have a vested interest as well, since having a large project like Angular as a user increases robustness and contributions.

Browsers

Leading web browsers also have a minimal stake in frameworks like Angular. They are incentivized to ensure that features utilized by these frameworks are well-supported and highly performant, in order to provide their users with the best possible experience. In fact, a lot of synthetic benchmarks test framework usage performance. For example, Google Chrome, Mozilla Firefox, macOS Safari, Microsoft Edge, and Opera.

Power Grid

We identified the interest and power of actors described in the stakeholders analysis. The most powerful actor with the highest interest is Google, which uses Angular for many of its products and is maintaining the Angular core team. Competitors have high interest in Angular project as their popularity is correlated with Angular success or failure, however they do not exert any significant power over it. Since Angular is written in TypeScript, which in turn heavily relies on JavaScript, the language specifications also have some power over the project, but with little to no interest. While web browsers are the main environment where Angular is used, they could have some power over the framework; however because browser vendors mostly only implement language and environment standards and experiment with early versions, they are not actually powerful. Community Contributors do have power over the project as their contribution can introduce new features or change functionality - their input is usually taken into account, however there is no way they can force Angular to make significant changes without substantial support. Web developers have almost no power over the project, although they can suggest changes that best suit their needs - which can then be picked up by community contributors and merged by the Angular Core Team. The Core Team exerts the most influence over the project's course and make decisions on features that are developed by them and merged from community developers.



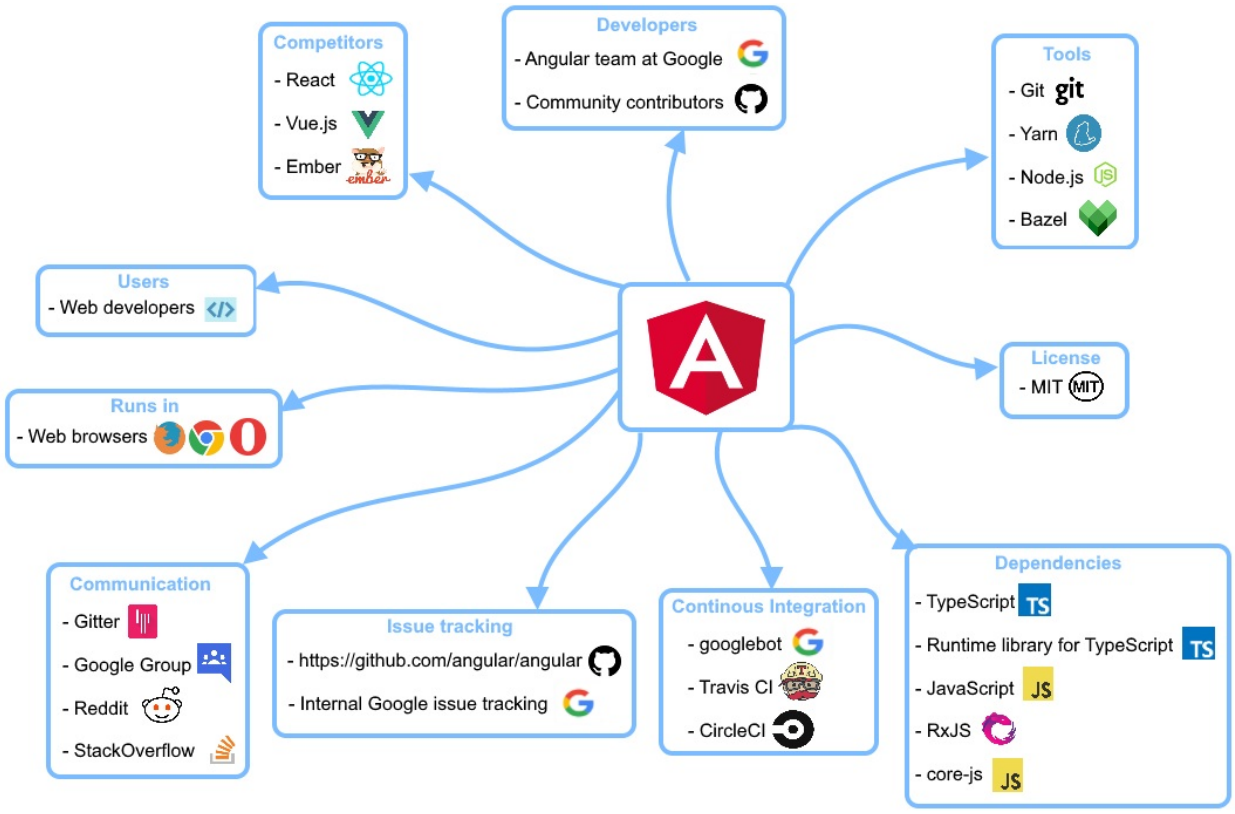
Context View

A context view describes the relationships, dependencies, and the interactions between the system and its environment. This view is relevant to understand the system's architecture, as it defines the boundaries of the system and how it interacts with external entities across all spectrums.

System scope & responsibilities

Angular is a front-end web framework. As a result, its main objective is to help developers build fast, responsive, robust user-facing web applications. Frameworks focus on establishing patterns that help with the programming and layout of the contents of web applications, by forcing the user to conform to its own opinionated designs (in contrast to how libraries work by providing just an interface to use). Technologies used by those frameworks include HTML and CSS for creating and laying out views along with JavaScript, TypeScript and, recently, WebAssembly for handling front-end logic such interacting with view or fetching data from server. Angular combines declarative templates, dependency injection, and tooling to make development quicker and easier to maintain.

Context View Diagram



Using the above context-view, from a high-level perspective, we can see that while the project itself is open-source, Google is the chief maintainer, and so we can see high reliance on Google tooling, especially for development. Other individual entities/groups are described individually below.

External Entities & Interfaces

- **Developers** - Angular is developed primarily by the dedicated team at Google (Angular Core Team) with contributors from the large open source community. They are responsible for developing, releasing, and maintaining the framework.
- **Users** - Angular is a front-end web application framework which means it is used by web developers to create web applications, and according to the StackOverflow it is one of 3 most popular ones as of 2018. Its users include individual developers, and companies creating web applications for their clients and creators of reusable components for Angular.
- **Runs in** - As Angular is a web framework it runs in web browsers - including almost all modern browsers for both desktop and mobile devices. A lot of consideration is given to make Angular work on as many platforms with little effort from the user.

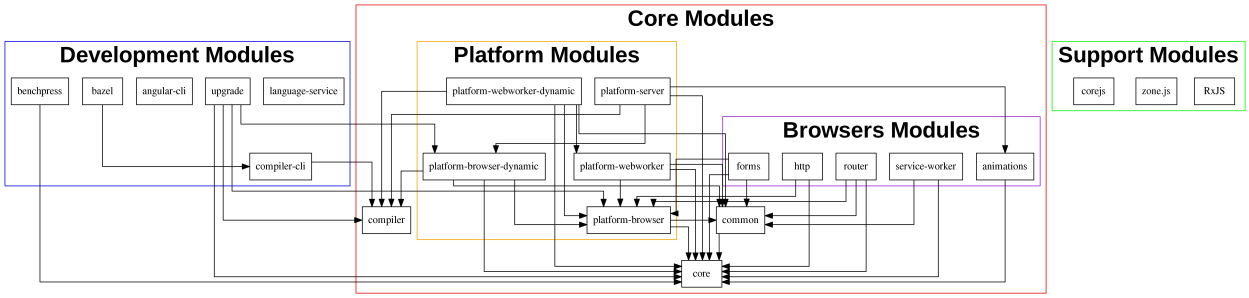
- **License** - Angular is an open source project, released under MIT license. It states that users have unlimited rights to use, copy, modify and distribute original/modified version of software.
- **Competitors** - Angular's competitors include all other web front-end frameworks. Providing an exhaustive list of them is challenging given dynamic development of current web ecosystem, but the most notable ones include React, Vue, Ember, and Blazor.
- **Tools** - Angular is using numerous tools for development. The project is using Git SCM as its version control software, Yarn as the dependency manager, Node.js as the JavaScript runtime, and Bazel as the build tool.
- **Dependencies** - Angular is developed in TypeScript and also heavily depends on numerous JavaScript and TypeScript libraries.
- **Continuous Integration** - For its CI and automation needs, the Angular team use Travis CI, CircleCI, and internal Google Bots. Those bots are used for example to check if a Contributor License Agreement was signed or to test if changes doesn't break internal Google repositories.
- **Issue Tracking** - Angular is using GitHub for its issue tracking, feature planning, and roadmap, though unfortunately, the core team seem to be using internal Google tools and keeping discussions offline, which means these are not accessible to open source contributors.
- **Communication** - Angular's official communication channels are Gitter and Google Group. Its unofficial channels are Reddit and StackOverflow, where users can reach out for help from the community.

Development View

The development view for a system describes the architecture related to the software's development process. As such, we have explained module organization, common design processes for components like code style, testing, logging, and building, along with an explanation of the codeline organization. Unfortunately, there does not exist any canonical documentation detailing the architecture of Angular for developers, so instead we have attempted to derive a relevant overview.

Module Structure

Angular source code is organized in to several modules that encapsulate specific functionality of the project, represented as npm packages which allow for easy installation by tools such as npm or yarn. These packages are usually referred to as `@angular/*` and are called "components" within Angular's documentation. Below, we present Angular's module organization diagram with their dependencies, which can be split into 3 groups following the convention introduced by the Angular team - Core, Development and Support modules.



Core Modules

Core modules provide functionalities that are used across whole Angular system and contain modules necessary for it to work.

1. **common** - The commonly needed services, pipes, and directives provided by the Angular team.
2. **core** - Critical runtime parts of the framework needed by every application - metadata decorators, Component, Directive, dependency injection, lifecycle hooks.
3. **compiler** - Angular's Template Compiler - it understands templates and can convert them to code that makes the application run and render.

Platform Modules

Platform modules provide functionalities that are platform specific. In the case of Angular those dependent on the browser itself and sometimes include polyfills. Those modules responsibilities include manipulating DOM (Document Object Model), threading and server side rendering.

1. **platform-browser** - Everything DOM and browser related, especially the pieces that help render into the DOM.
2. **platform-browser-dynamic** - Includes providers and methods to compile and run the app on the client using the JIT compiler.
3. **platform-webworker** - Angular's support for threading and background calculations using web workers.
4. **platform-webworker-dynamic** - It contains JIT compiler specific options for **platform-webworker** module.
5. **platform-server** - Angular's support for server side rendering.

Browser Modules

Browser modules provide functionalities that are used to perform various actions in the web applications like changing pages via routing, making HTTP calls, or animations.

1. **router** - The router module navigates among web application pages when the browser URL changes.
2. **http** - Angular's old, soon-to-be-deprecated, HTTP client, being being replaced by '@angular/common/http'. The HTTP client is used for sending and retrieving resources over internet using HTTP protocol,.
3. **forms** - Both template-driven and reactive forms allow users to interact with web application's logic such as log in, placing order or booking flights.
4. **animations** - Angular's animations library for applying animation effects such as page and list transitions.
5. **service-worker** - a network proxy script that manages caching for an application. Angular's service worker is designed to optimize the end user experience of using an application over a slow or unreliable network connection.

Development Modules

Development modules provide functionalities for developers creating web applications using Angular as front-end. They are **not** necessary for Angular to work but contribute to good development experience.

1. **language-service** - The Angular language service analyses component templates and provides type and error information that TypeScript-aware editors can use to improve the developer's experience.
2. **upgrade** - Set of utilities for upgrading AngularJS applications to Angular.
3. **compiler-cli** - The Angular compiler CLI tools, which are mainly used for compiling templates.
4. **angular-cli** - Which lies outside of main Angular repository. It contains Angular CLI tools, which consist of generating scaffolding for web applications, components or modules among others.
5. **benchpress** - Angular's performance measuring tool. Developers can benchmark their applications using this module to ensure best performance of their applications.
6. **bazel** - Angular's support tools for building Angular applications with Bazel. This still work in progress and Angular Core Team is transitioning from shell scripting build system.

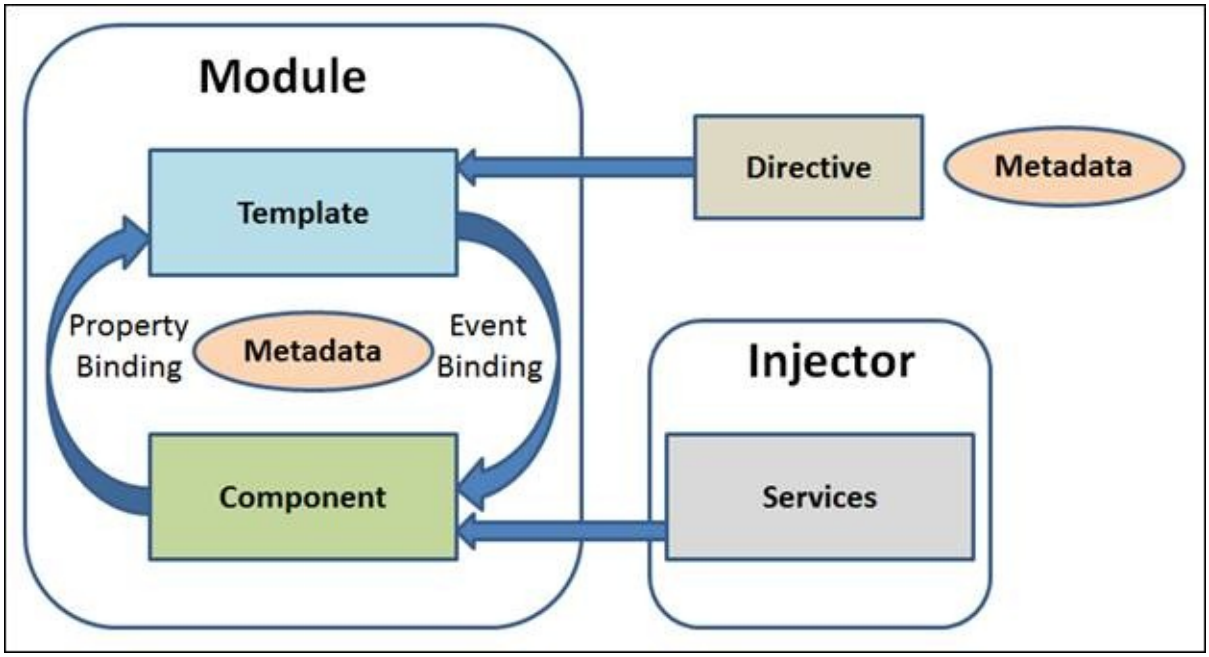
Support Modules

Support modules are used by the Angular project internally but are not actively maintained in the project itself. Those mostly include JavaScript and TypeScript libraries.

1. **RxJS** - Many Angular APIs return observables. RxJS is a library for reactive programming using Observables, to make it easier to compose asynchronous or callback-based code.
2. **zone.js** - Angular relies on zone.js to run Angular's change detection processes when native JavaScript operations raise events.
3. **core-js** - Polyfills plug gaps in a browser's JavaScript implementation. `core-js` is a popular library for that.

Common Design Model

Common Patterns



In the above diagram, we can see an overview of an Angular application's architecture. Modules define compilation context, and contain Templates which contain the view layout while the Components contain the business logic. Metadata, or state, is shared and Property and Event Binding are used to keep both these parts in sync. Services provide specific functionalities, and are provided to the application via the Dependency Injector. Directives provide view logic to the Templates.

The Angular project utilizes the industry-standard pattern of Object Oriented Programming, more specifically Class-based Programming. Each class' behavior is defined as a blueprint and objects are instantiated explicitly based on a class. Further, this allows for inheritance, where you can define common behaviors in the parent and extended behavior in the subclasses. Logic is encapsulated within the class definition and it is good practice for objects of that type to utilize specific methods that change data values and attributes.

Angular also utilizes the paradigm of Dependency Injection to provide components with their dependencies at instantiation time. Dependency Injection (DI) is a way to create objects that depend on other objects. A DI system supplies the dependent objects, i.e. the dependencies, when it creates an instance of an object. The primary advantage of DI is that it forces each file to clearly state its dependencies, which makes for a clear separation of concerns resulting in easier debugging and testing. A minor disadvantage is that code ends up being more verbose and clunky.

Common Processing

Common processing approaches address aspects of the architecture that require a standard approach across the whole system. This is a key task as it contributes to the overall technical coherence of the system and clarifies how and where processing is done.

Modularity

The Angular project contains multiple directories like `aio` for Angular.io, `docs` for Documentation, `integration` for the End-to-End integration tests, `packages` for the core sub-modules, `scripts` for project scripts, and `tools` for tooling.

Message Logging

The logger is instantiated only once - during the bootstrap step. All logging is done through this common, app-level logger instance. The advantage of this is that all logs are handled in a universal manner with uniform formatting and styles. Further, logging is enabled only in development mode and not in production to prevent exposure of sensitive data to the user's console, and in the case of a framework like Angular, to avoid spamming the actual applications' logs. Angular does not log much in general, rather it restricts itself to deprecation warnings and framework errors.

Testing

All tests are run in parallel, to decrease testing time and increase throughput. A side-effect of this is that tests do not rely on lack of race-conditions or undefined dependencies to work correctly. Any tests that are dependent on other components are required to create mocks or instantiate dependencies as necessary.

Standard Design

Standard design approaches are used when designing the system's various elements, and only start to emerge while subsystems are fleshed out.

Coding Style

Google has standardized rules for all their JavaScript code in the JS style guide and all contributions are expected to follow it strictly. Each and every contribution is tested against these guidelines by the CI tooling and is rejected until changes are made to conform. This is done to keep the codebase uniform and remove any comprehension overhead for developers.

Commit Style

The Angular project has strict rules about git commit messages as described here in order to ensure consistency and clarity.

Internationalization

Angular as a framework is written in TypeScript, which means that the coding language is English. However, the website with documentation is available in multiple languages, often maintained by the community. Official translations are maintained on GitHub in language-specific locations, i.e. English and Chinese.

Standard Software Components

This refers to what common software should be used for different aspects and how it should be used to maintain consistency. These are usually the result of making higher-level decisions or identifying reusable components.

Third Party Libraries

3rd party libraries are used to provide a consistent and stable API for the Angular project to use. This reduces development time and effort, as common patterns can be offloaded to these libraries. While this does open up possible security concerns, it is an industry-wide practice to utilize libraries that provide stable, tested functionality. A comprehensive list of these dependencies can be found in the package.json.

Technical Debt

Technical debt is a gap between creating a perfect solution: adhering to the architectural design, using good programming standards, with proper documentation and thorough tests, and creating a working solution: as quickly and cheaply as possible. Technical debt is an essence those aspect of software that are incomplete, immature, or inadequate *(Cunningham, 1992)* and are imposing a risk of causing problems in the future if not properly fixed.

We analyzed the Angular project quantitatively (static code analysis) using tools such as Codebeat and qualitatively by manual inspection to assess evolution of technical debt. While we have analyzed the project to the best of our abilities, it is possible that there may be minor details we missed due to the complete lack of documentation.

Historical Analysis

The Angular project is sort of an evolution from Angular 1, built with performance and maintainability in mind. It was a complete rewrite, in terms of code and architecture, which was initially performed privately within Google. Afterwards, Angular 2 was announced publicly as a beta software in December 2015.

Angular now uses semantic versioning and as a result, only major version changes can introduce breaking changes while minor version changes should handle bug/security fixes. This is a welcome change, as inconsistent API deprecation can lead to technical debt, and was a major point of contention in the community when Angular (2) was announced and required a complete rewrite of AngularJS (1)

applications. Angular version 3 was skipped to prevent numbering confusion, and since then, Angular has matured with version 6 currently in beta and version 5 considered stable.

It is not practically feasible to analyze the entire history of the Angular codebase, especially considering the lack of documentation. Instead, we opted to choose an example of debt evolution - specifically, testing and building debt. This aspect is quite interesting as the Angular project is currently in the middle of a major rewrite of their testing and building architecture. While it was earlier handled by monolithic build and test shell scripts, they are now going to be handled with a Google-created tool called Bazel. This is a very important change that effectively removes a significant portion of technical debt from the project. We have discussed more about this change in our Solutions in Motion. Furthermore, we have touched upon other specific aspects of historical debt in the individual sections of this chapter.

Codebase Analysis

We analyzed the repository using SonarQube, which is designed to find potential bugs, vulnerabilities, code smells, coverage, and code duplication. This analysis revealed that Angular has amassed 348 days of technical debt according to SonarQube, which gives it the highest *A* rating. The KPIs identified are:

- **Bugs** - SonarQube identified 5360 potential bugs which resulted in the rating *D*, which means that there is at least one critical bug identified. It predicts the remediation effort to be 49 days.
- **Vulnerabilities** - SonarQube identified 6 possible security issues, with rating *B*, meaning at least one minor vulnerability and remediation effort of 0.5 hours.
- **Code Smells** - SonarQube identified 23k (thousands) of code smells, most of them not critical. This contributes to 347 days of effort from 348 days identified. This is because bugs and vulnerabilities are ignored in the debt analysis, as they are different category of problems.
- **Code Duplication** - SonarQube identified 7168 of blocks, 201544 lines and 1488 files duplication, with the density of 6.6%. This ratio is quite high and it appears that these duplication aren't taken into account while calculating effort required to pay the technical debt.

This analysis finds that Angular project hasn't amassed significant technical debt in any unsolvable ways. The identified debt instances are not critical and are few and far between. This can be caused by a few different reasons:

1. TypeScript offers typed code and convenient syntactical sugar which can help alleviate the buildup of technical debt. This is opposed to JavaScript which, without rigorous code reviews, is known in the community for being cumbersome.
2. Fast release cycles for Angular - on a 6 month schedule.
3. Complete rewrite of an existing product along with a switch in programming language kickstarted the development in the right direction while avoiding common pitfalls.
4. Famous code quality standards of Google, but this might just be wishful thinking on our part.

It is possible that some technical debt instances are present in the project but the tools and procedures we used might have missed them. It is however unlikely that this is a large amount as otherwise they would be easily noticeable.

Testing Debt

Building on our previous analysis of Angular's testing tools, we proceeded to observe the output of each testing suite and continuous integration tool, to understand their purpose and utility. We also analyzed what sort of testing debt they contribute to the project.

Angular uses Jasmine to unit and function test each component with a clear definition of the behavior of the test case itself. Karma is a "test-runner" environment for JS, which runs suites of tests in multiple real browsers and devices, i.e. ensuring cross-compatibility and preventing regressions. It is a crucial tool for frameworks like Angular, which aim for maximum compatibility leading to widespread usage. Protractor is an end-to-end (e2e) testing library built by and for Angular. It support tests written in `Jasmine`, hence decreasing the overhead of having multiple test definition frameworks. Protractor further uses WebdriverJS which uses native events and browser-specific drivers to simulate real-world user interactions for testing. This helps catch bugs that might not be apparent in unit tests. End to end tests are crucial in this regard and further guarantee the validity of the code. BrowserStack and SauceLabs are cloud-based, cross-browser and cross-device testing tools. They do not offer any specific testing frameworks or definitions, rather they are Continuous Integration tools for running tests through `Karma`, as explained earlier. Angular uses BrowserStack and SauceLabs to track cross-compatibility and continually test code against the test suite in a public, online manner. The result of this integration can be seen in this

compatibility matrix for the `master` branch. Here, we can observe another aspect of testing debt - it appears that tests are failing for `Internet Explorer 7`, `Android < 6.0`, `iOS 10.x`, and `Edge on Windows 10`. Only `Chrome` and `Firefox` seem to have maximum support. Angular currently uses a monolithic test and build shell script, which is in the process of being replaced by Bazel. We have detailed this technical debt mitigation in Things in Motion.

Things in Motion

Bazel is a fast and correct build and test tool, built with concepts like caching, incremental builds, dependency analysis, and parallel execution in mind. Bazel was built by Google, and is now developed by a core Google-employed team and various community contributors on Github, similar to how Angular is managed. Fabian Wiles directed us to his PR #909, where he started the implementation of Bazel in Angular. We analyzed the changes made within and made some observations:

- Bazel does NOT yet have documentation for Javascript/Typescript projects. This is current technical debt, and is significant as the migration is already happening, but non-core-members are not aware of how Bazel works in the first place.
- As mentioned earlier, core member are discussing technical implementation details offline without documenting their ideas anywhere. Moving forward, this can contribute to technical debt, through the bus factor, as other contributors are not made aware of key points.
- Angular uses the `ngc`, i.e. Angular compiler tool and there is a slight compatibility issue with Bazel and 3rd party libraries. Hence, a workaround has been used for now. This is an obvious starting point for potential technical debt, until the issue is resolved and the workaround is removed.

Conclusion

This chapter summarized our analysis of the Angular project. We described stakeholders and provided contextual analyses of the project's ecosystem. We concluded that Angular is primarily Google's project and hence they control its features and releases. Our contact with Angular Core Team from Google was successful which proved that project is open for open source contributions, however finding useful documentation is not straight-forward. We investigated project modularity, common design patterns, coding styles, and the tools that are used. We found that there are rather strict coding rules that contributors need to adhere to. There are numerous automation tools used during development, testing, and releasing of the system. Our analysis of technical debt found that it is rather minor, in part thanks to the complete rewrite performed by Google from AngularJS. Furthermore, Angular is released regularly in a 6 months cycle and in the next one, Angular 6, will introduce the new building system based on Bazel, along with a new render engine which increases performance and reduces size of applications. Those changes are accompanied by an updated Angular component compiler for Custom Elements and a CLI upgrade. Our findings lead us to believe that Angular will continue to lead front-end frameworks popularity rankings for the near future.

References

1. Nick Rozanski and Eoin Woods. 2011. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley Professional.
2. Angular - Angular - Contributors. https://angular.io/about?group=Angular
3. Ian Allen. 11-01-2018. https://stackoverflow.blog/2018/01/11/brutal-lifecycle-javascript-frameworks/
4. AngularJS - Superheroice JavaScript MVw Framework. https://angularjs.org/
5. Google Javascript Style Guide. https://google.github.io/styleguide/jsguide.html
6. Angular - Angular Dependency Injection. https://angular.io/guide/dependency-injection
7. angular/angular: One framework. Mobile & Desktop. https://github.com/angular/angular
8. TypeScript - JavaScript that scales. https://www.typescriptlang.org/

Docker SwarmKit

Dhruv Batheja, **Chantal Olieman**, **Sven Popping**, **Vincent Robbemond**

Abstract

SwarmKit is a toolkit for container orchestration, with a focus on security and scalability of distributed systems. The project is managed by Docker Inc., is just over 2 years old and has been open-source for almost 2 years. SwarmKit is one of the main container orchestration tookits on the market today, and is being actively developed. This is an analysis of the SwarmKit architecture, by exploring the full project, its stakeholders, developer collaboration, and technical debt. First, a stakeholder analysis along with a context view are discussed, followed by insights from a core SwarmKit developer. After that, differrent viewpoints are discussed for the SwarmKit project in order to create a better understanding of the underlying architecture.

Table of Contents

Introduction

SwarmKit is a Docker initiative that began in early 2016, and is a toolkit for orchestrating distributed systems. It includes primitives for node discovery, raft-based consensus, task scheduling etc. SwarmKit is the orchestration component in the Moby Project, and covers Cluster management and orchestration features in Docker Engine 1.12 or later. It uses the Apache License for use, reproduction, and distribution.

The main collaborators of this project are engineers from Docker Inc., with small contributions from the community. SwarmKit runs on all popular server environments that most of the popular cloud-providers provide. The project is very active and has grown tremendously since its inception. There are daily commits to the repository and issues are actively being created and closed. The releases of SwarmKit are currently coupled with Docker releases. The most recent stable release of SwarmKit is `v1.12.0`.

This chapter is written as part of the Delft Students on Software Architecture book 2018. This chapter aims to summarize the architecture of SwarmKit, provide details about the stakeholders, present different views ranging from development to functional, and evolution of SwarmKit. Hence, this chapter aims to cover not only the technical aspects of the project, but also provide a broader view. This chapter also covers our interaction with a core-developer from SwarmKit team, as well as contributions made. This also includes a section about possible scopes of improvement that are discussed in the Technical debt section.

Stakeholders

The definition of a stakeholder is: people, organizations, groups and/or companies that have interest in the realisation of a project. Stakeholders can be affected by or affect the actions, objectives and policies of the organization.

Rozanski and Woods classification

In the book "Software Systems Architecture" of Rozanski and Woods [3], 11 stakeholder types are described. Below are the types of stakeholders that apply to SwarmKit.

Acquires

The acquirer of SwarmKit is Docker Inc. They employ the core team of SwarmKit. This core team is in control of the roadmap. They have a daily stand-up meeting in which they discuss the issues they want to fix that day.

Assessors

There is no clear role of an assessor in this project. Docker SwarmKit is licensed under the Apache 2.0 license, which means everybody is allowed to use the software, without warranty or liability from Docker Inc. Every pull request is checked by two or more members of the core team. They also asses the conformance to standards and legal regulations.

Communicators

The most important communicator for SwarmKit is the team itself, they regularly monitor developer platforms like Docker Forum, Google Development Group and Stack Overflow for questions from users/contributors.

Besides the developers, Docker Inc. also collaborates with so called training partners. These are companies certified by Docker which provide education courses.

Annually Docker hosts Docker Con, last year two team members of SwarmKit gave a presentation titled: Under the Hood with Docker Swarm Mode.

Developers/Maintainers

The MAINTAINERS file on the repository contains a list of all the maintainers of the project. Beside the core team there are a few active developers that contribute to the system. Most tasks like bug fixes, enchantments, etc. are fixed by the core team.

Suppliers

SwarmKit is written in Go, and compiled binaries run on any Unix based OS. GitHub provides the repository where the code is stored. Resellers are companies that offer a service to this customer that use the Docker Enterprise platform as core.

Support staff

Docker Inc. itself handles all support. On the website https://support.docker.com you can post tickets/questions regarding all Docker systems.

Testers

The developers create unit/integration tests, these tests are run by CircleCI and after, CodeCov will create a coverage report.

Users

SwarmKit is used by the enterprise customers of Docker. SwarmKit is also used by other companies and private users.

Other classification

Listed below are additional stakeholders which go beyond the the classification of Rozanski and Woods.

Media

A few websites have posted articles about SwarmKit, describing SwarmKit and how it can improve organizations and help with scalability. These may persuade more organizations to use SwarmKit as bases for their infrastructure.

Researchers

A article has been published on improvements of the container scheduling for Docker using Ant Colony Optimization [1].

Partners

Docker Inc. has five Spotlight Partners: Alibaba, Cisco, HP, IBM and Microsoft. These partners strive to improve the user experience for developers, by creating a hybrid infrastructure that supports continuous delivery of applications and services. All of these functionalities should be available across all major operating systems like Windows and Linux.

Power vs. interest grid

A method to classify stakeholders is by using Mendelow's power vs. interest grid [2]. Figure 1 shows the power vs. interest grid for the Docker SwarmKit project. There are four categories: minimal effort, keep informed, keep satisfied and key stakeholders.

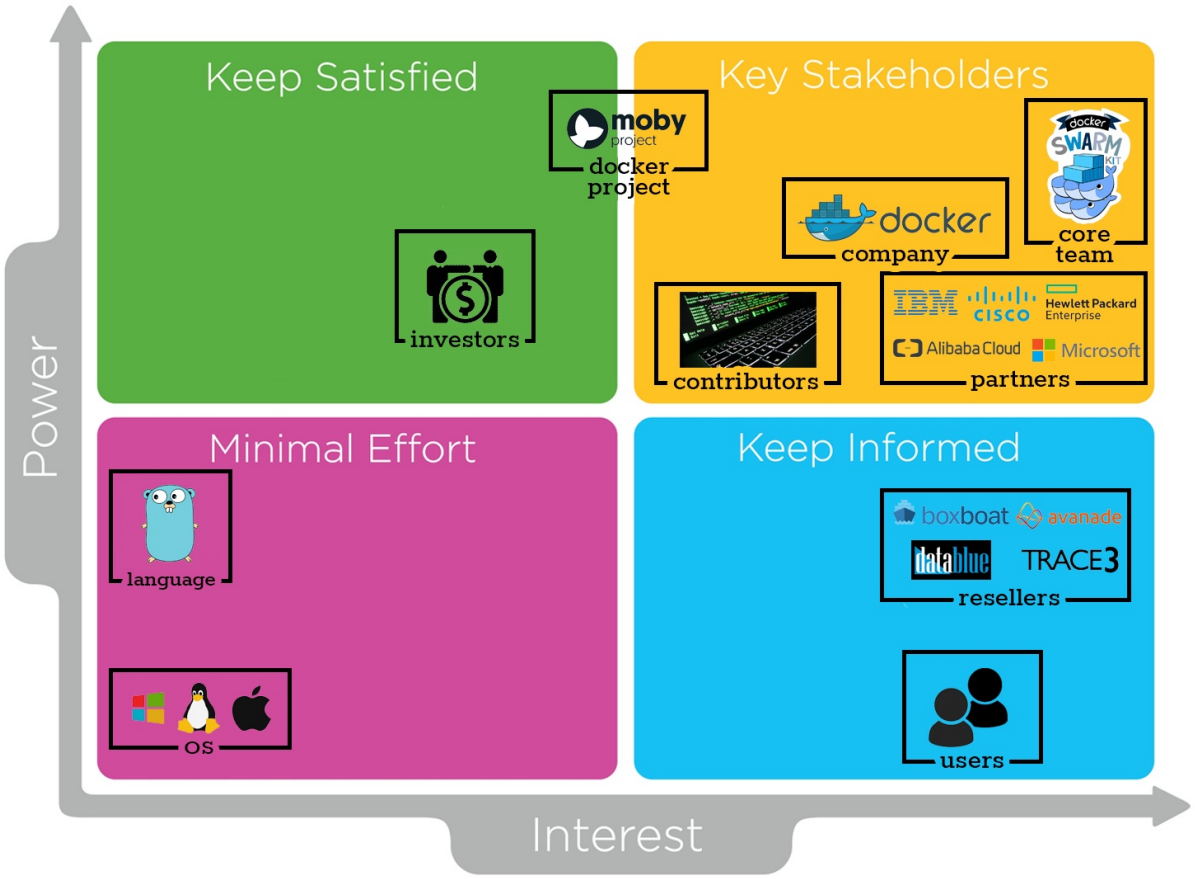
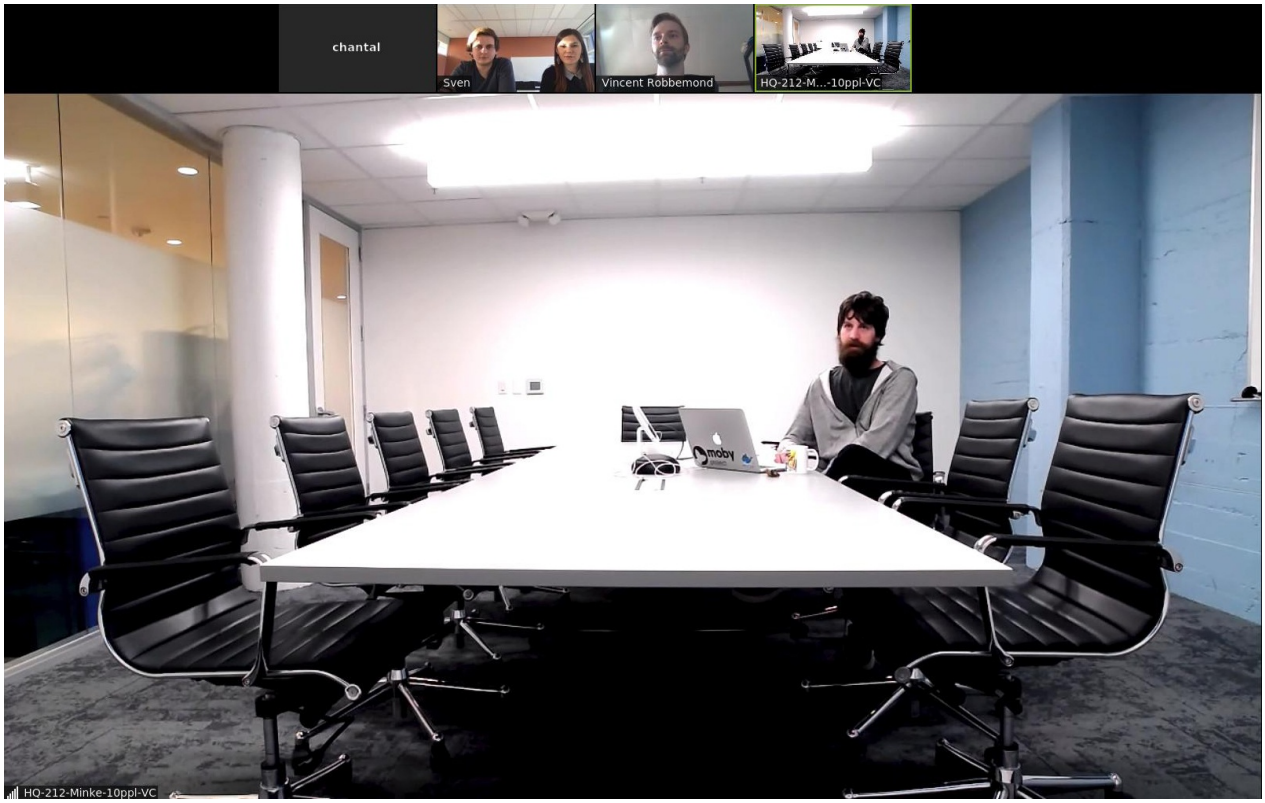


Figure 1: Power vs. interest grid for the Docker SwarmKit project

Integrator insight

We scheduled a video call with @dperny, a core team member of Docker SwarmKit. Drew Erny graduated in 2016 at the University of Alabama, and has been working at Docker for 2 years.



During this call we asked about the allocator (docker/swarmkit#2516), which is one of the big tasks he is working on. He explained the core issue is there is no difference between initialization and allocation of resources. The first version was written by one developer, who was fluent in C, but fairly new to Golang. Therefore the code is messy. Currently Drew is working on docker/swarmkit#2579 as the first PR in a series of PRs rewriting the allocator. Due to a lack of allocator-related tests, the whole rewrite has to be manually tested. As Drew described, the exact architecture of the allocator is not decided beforehand, but formed throughout the process of rewriting. This makes sense considering in the process of rewriting, the developers' understanding of this code grows and therefore their view on the architecture may change as well.

The development of SwarmKit is currently following a downward trend in terms of new features in the roadmap. Most open issues are bugs that were reported by enterprise customers. Debugging these issues can be a hard task, as many customers do not give direct access to their clusters. These customer escalations can take up quite some time, depending on the severity of the issue. According to Drew, some issues can take up the entire week, leaving no time for developers to work on other tasks.

Of the few SwarmKit contributors, most work at Docker. Therefore a lot of communication happens internally, face-to-face or through Slack. This makes it easy to 'forget' SwarmKit is an open-source project, and communication towards the community can be improved.

A lot of files in the project contain long methods which is a violation of the SOLID principle. However the team argues that if the function interface is good and testable and the code is dry (no repetition), then long methods are not directly seen as a problem. Secondly splitting these methods up into different function may obscure the complexity of the main method.

Upon hearing our technical debt analysis, Drew agrees that the project contains severe testing debt. Apart from a lack of functional tests, the project also lacks performance tests. The team is able to identify when a cluster is broken, but can't yet figure out what causes the instability.

In the near future, the SwarmKit team aims to fix most of the technical debt by improving code quality and rewriting parts of the code that cause a lot of bugs. One of the long term goals is to integrate SwarmKit with Kubernetes as Docker currently support Kubernetes.

Context View

This section describes the context view of the `docker/swarmkit` project. The context view portrays the relationships and interactions of `docker/swarmkit` with the environment.

System Scope

The project repository of SwarmKit uses the following description: *"SwarmKit is a toolkit for orchestrating distributed systems at any scale. Machines running SwarmKit can be grouped together in order to form a Swarm, coordinating tasks with each other."*

SwarmKit is a separate project in the Docker project. It focuses on implementation of the orchestration layer and is used as is in the main project hence efficiently separates concerns.

Context Model

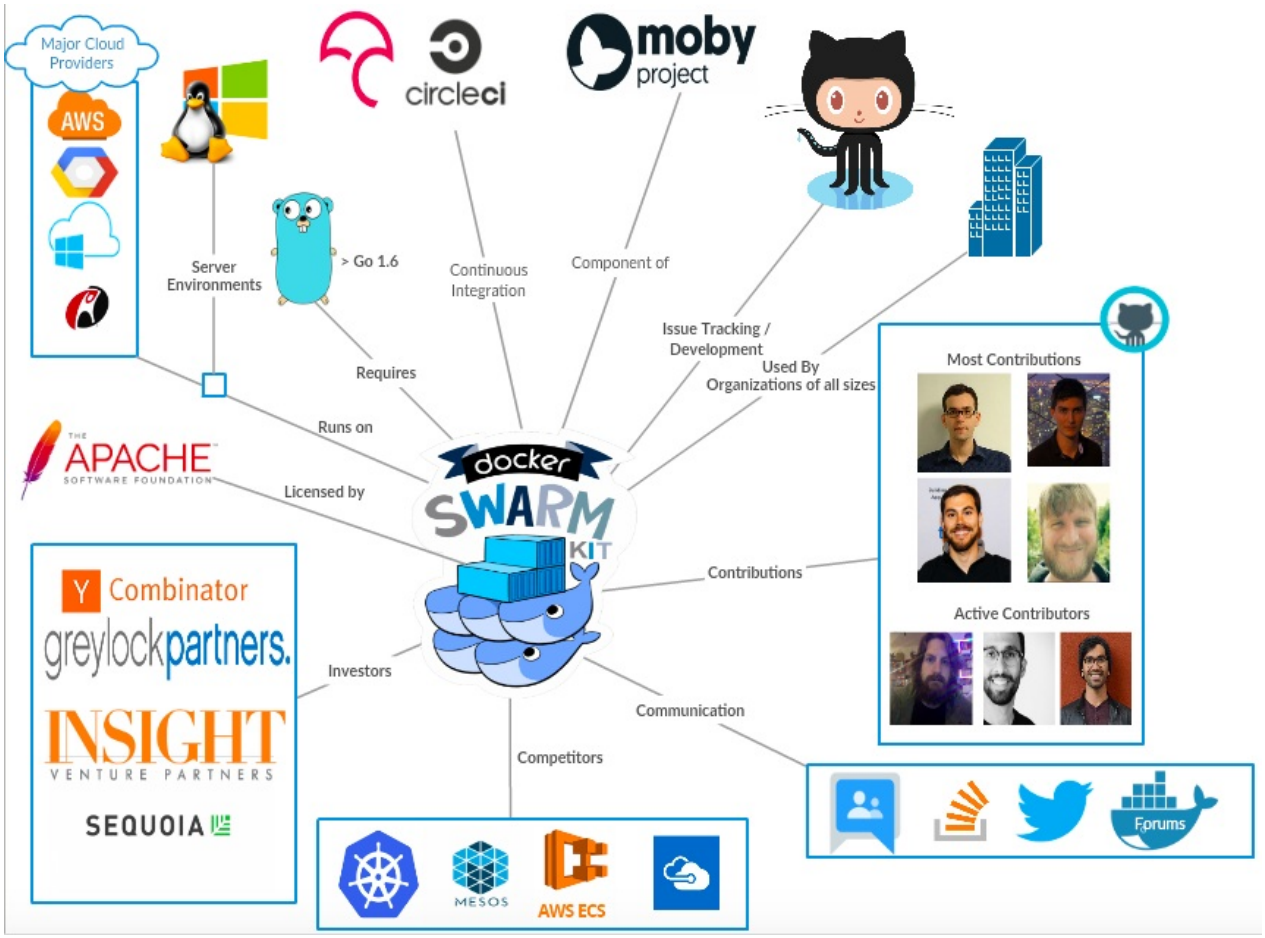


Figure: Swarmkit Context Model

The figure shows a simple black-box context model implementation of the `docker/swarmkit` project. A small description of the important entities in the diagram is as follows:

- **Overview:** SwarmKit is the implementation of the orchestration layer built for the Docker project and aims to abstract the features like Orchestration, Scheduling, Cluster Management and Security.
- **Requires:** The project is implemented in the Go programming language and requires **Go version 1.6 or greater**. The project also requires **Protobuf 3.x or higher** to regenerate protocol buffer.
- **Supported on:** Thanks to the multi platform support of Docker, SwarmKit is supported on all operating systems.
- **Integrations:** The project uses integrations like CircleCI for continuous integration and Codecov.io for gaining insights about the code coverage for testing the impact of new changes made in the system.
- **Inspiration:** Moby is an open-source project started by Docker to enable and accelerate software containerization. SwarmKit is the orchestration component in the Moby project. The architectural decisions for SwarmKit are heavily influenced by Moby guidelines.
- **Community:** The project only uses github for issue-tracking and development collaboration. It has an active community support and adoption. The contributors can be found in the MAINTAINERS.

- **Communication:** This sub-project does not have a big online presence in social media, but the Docker project can be reached out to via several communication mechanisms like: the Docker forums, IRC, Google Group, Twitter and it also has a Stack Overflow tag.
- **Sponsors:** The Docker project has raised over 242M from 24 investors in 10 rounds of funding and is valued over 1B. It was incubated by YCombinator and the lead investors include Sequoia Capital, Greylock Partners and Insight Ventures.
- **Competitors:** The main competitors of this swarm orchestration project include Kubernetes, Mesos, Amazon's Elastic Container Service, Azure's container service (AKS).
- **License:** The project is licensed under *Apache License 2.0* to define terms and conditions for use, reproduction and distribution.
- **External and Internal Entities:** External entities where SwarmKit relies on are the cloud providers on which SwarmKit is deployed, the Go programming language in which it is written, CI tools CodeCov and CircleCI and the communication tools used by the core team.The investors and competitors shown in the context diagram can all be categorized as external entities as well. Internal entities include the developers as most of the contributors are employees of Docker and the Moby project as it is a Docker project.

Development view

Common Processing Model

Encapsulating common processing across sections of the project into separate code units contributes to the overall coherence of the system and tackles issues like duplication. Since SwarmKit is written in Go, it makes use of the excellent package management that Go provides. Moreover, Go enforces a strict import regime which makes the code more readable and less prone to unnecessary import clashes.

- **Common data store:** SwarmKit's data store is built on top of go-memdb, which is a in-memory database built on immutable radix trees. SwarmKit also uses Bolt to provide a simple, fast, and reliable database.
- **Third party libraries:** SwarmKit uses the Cobra for its CLI module. It also uses Cloudflare's CFSSL for signing, verifying, and bundling TLS certificates. Another notable dependency is containerd, to manage the complete container lifecycle of its host system.
- **Logging:** SwarmKit uses Logrus for logging.
- **Scheduling & Batching:** SwarmKit's scheduling algorithm works by balancing the number of task over the nodes in the cluster. However replicas should not be on the same node, when possible, to prevent a single point of failure.
- **Orchestration:** A service gets translated into a number of tasks, these are managed by the orchestrator. There are two types of events an orchestrator handles, service-level events and task-level events.
- **SwarmKit's Task Model:** A task is a "one-shot" execution unit. When a task fails it while never be executed again. The orchestrator may create a new task to retry.

Instrumentation Analysis

Instrumentation refers to the ability to monitor or measure the level of a product's performance, to diagnose errors and to write trace information.

As of now, SwarmKit doesn't have any specific instrumentation code. SwarmKit uses Logrus for all of the project's logging. SwarmKit uses Testify with features like *easy assertions*, *mocking*, *testing suite interfaces and functions*.

Developers can use instrumentation locally, for instance one can incorporate gmx to query the internal state of their Go application. Secondly, Delve, a full featured debugging tool for Go.

Standardization of design

For software developers to contribute to Docker swarmkit, some clear guidelines are set in CONTRIBUTING.md. A guide explains setting up a Docker development environment and the contribution process for all Docker projects. The community guidelines are mentioned specifically, these are general guidelines for the community as a whole and are enforced accordingly.

Code conventions

While reporting an issue, it is important the issue include the `docker version` and `docker info`. This information helps the team to review and fix the issue. One should also take care as to not submit duplicate issues.

When submitting a change, the changes should be made on a feature branch in a forked repository. The name of the branch should look like XXXX-something, where XXXX is the number of the corresponding issue.

When creating or modifying a feature, it is desired the documentation is updated accordingly, here also the style guide is referenced, containing instructions on building the documentation.

To keep all code clean, gofmt should be used to format all submitted code. Specifically, `gofmt -s -w file.go` should be run on every changed file.

PR's should always be rebased on top of master, without any other branches mixed into the PR. Commits should be squashed into logical units of work before making a PR and the description should contain the issue being addressed.

Overall, the coding guidelines from the Go community are followed.

Signing

Every commit should be signed-off, agreeing to the Developer certificate.

Standardization of testing

To ensure a reliable build process for SwarmKit, standard practices have been adopted by the team to test the adapted or added code.

Testing framework

SwarmKit leverages a standard project structure to work well with the standard Go testing framework. Tests are run with the `go test` command, which automatically executes any function of the form:

```
func TestXxx(*testing.T)
```

In line with the testing framework, all tests have a filename that end in `_test.go` and live in the same directory as the file being tested.

Automated testing

All code changes are tested automatically by triggering a test on CircleCI. The `README.md` in the main folder holds a badge showing result of the latest tests on the master branch: 

. NB: The result of the automated test build for our first Pull Request can be found here.

Module Structure

Concept of SwarmKit

SwarmKit can group hardware in order to form a *Swarm*, in which tasks can be coordinated among individual pieces of hardware. Once a machine joins, it becomes a node in the Swarm, called a *Swarm Node*. Figure 2 shows an example of a Swarm.

There are two types of Swarm Nodes, the worker node and the manager node. The worker node, called *Agent*, is responsible for running Tasks using an Executor. The manager node, called *Master*, accepts input from the user and is responsible for reconciling the desired state with the actual cluster state.

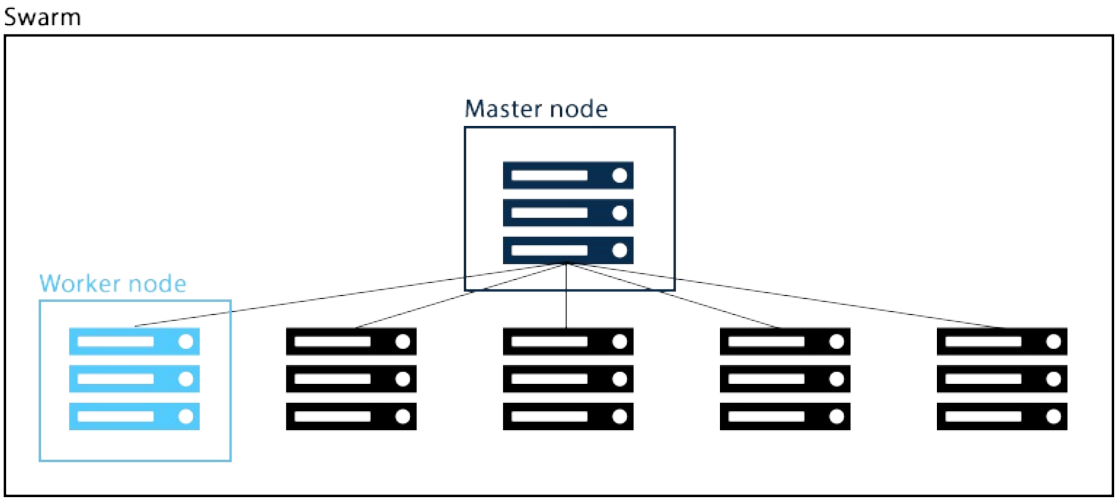


Figure 2 Example of the hardware structure of a Swarm

High level components

Figure 3 depicts the dependencies for Tasks, Services, Master and Agents. *SwarmCtl* is used to communicate with the cluster. The `swarmctl` command is used to to create, update and delete services and add/remove nodes from the swarm.

When a service is added to the cluster through the API, the Master will generate tasks, allocate resources, and pick an available node to execute the tasks. For the coordination of the tasks SwarmKit makes use of the Raft Consensus Algorithm.

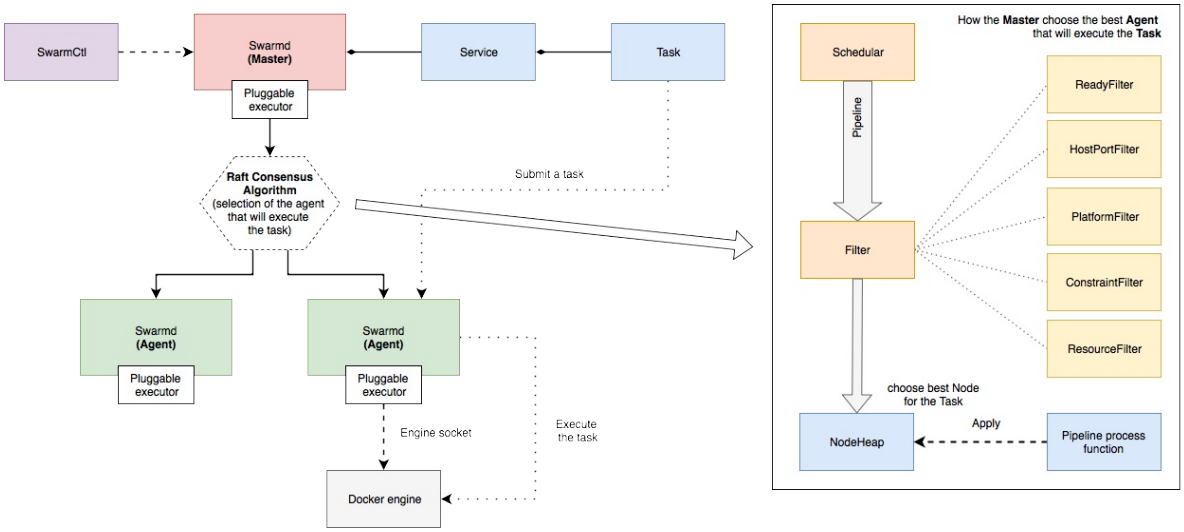


Figure 3: Dependencies of the Task, Service, Agent and Master.

Within the Master node there are four main parts, the Orchestrator, the Allocator, the Scheduler and the Dispatcher. The Orchestrator is there to ensure that the services have the appropriate set of tasks running in the cluster. The Allocator is responsible for dispensing the required amount of resources. The Scheduler assigns the tasks to the available nodes. The Dispatcher handles all communication between the Master and the Agent.

A worker consists of two parts, as shown in figure 3, the agent and the engine. The Engine runs the containers. The Agent coordinates the work and maintains the connection with the Master, the Agent notifies the Master of the current state of the assigned tasks.

Dependencies of components and models

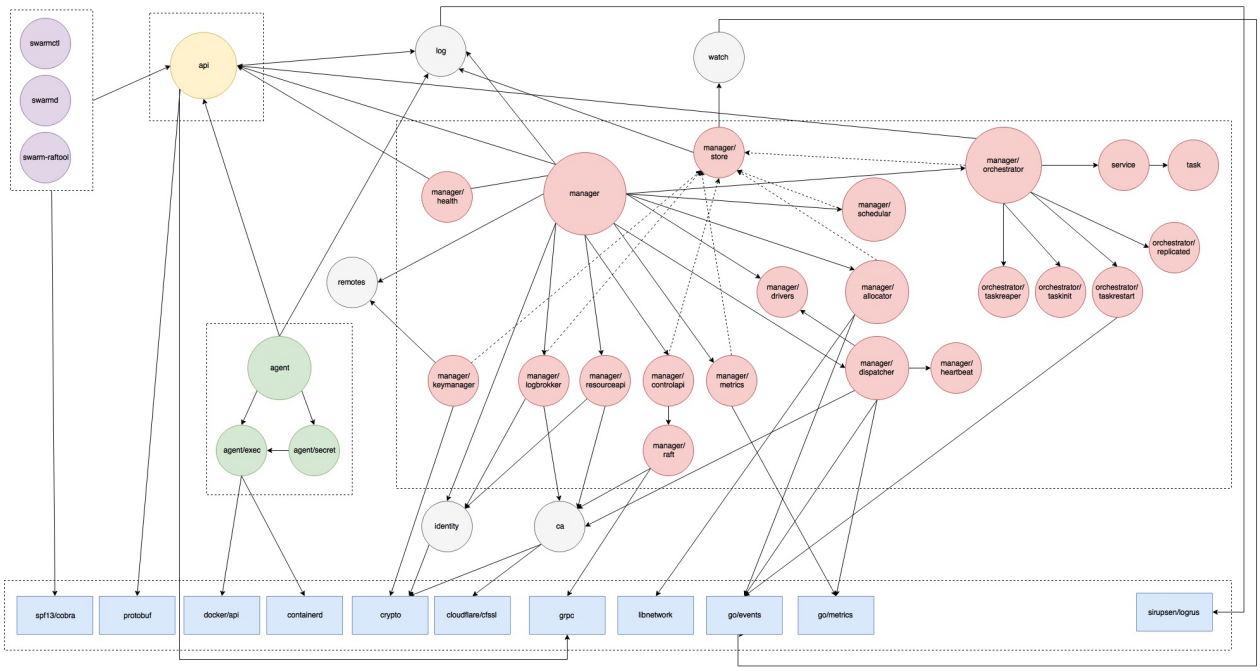


Figure 5: In-depth overview of the dependencies of components.

Figure 5 depicts the in-depth dependencies of the SwarmKit project. There are six types of components in this project, each type of component will be explained.

Blue components

Blue depicts vendor components, of which only the most important are shown. The vendors not shown in this figure are helper classes/functions. The full list of vendors can be found in `vendor.conf` .

Yellow components

Yellow depicts the API, which is based on protocol buffers. All communication between different components is event driven and go via the API. For readability, only the important connections are shown.

Purple components

Purple components are the available commands to interact with the SwarmKit cluster.

Grey components

The grey components are global helper classes/function. The `log` is the global logger of the system, `remotes` keeps track of remote addresses by weight, informed by observations, `identity` generates random ids which are used to identify components within the Swarm, `ca` is the certificate authenticator of SwarmKit and `watch` starts a stream that returns any changes to objects that match the specified selectors.

Green components

The green components are part of the agent module. The `agent/exec` is the executor for a container, currently the containers based on `dockerapi` and `containerd` are supported.

Red components

The red components are part of the manager, which looks like a very complex system.

The module `manager/store` stores all the information about the clusters. The module `manager/health` keeps track of the manager's health and reports it to the API.

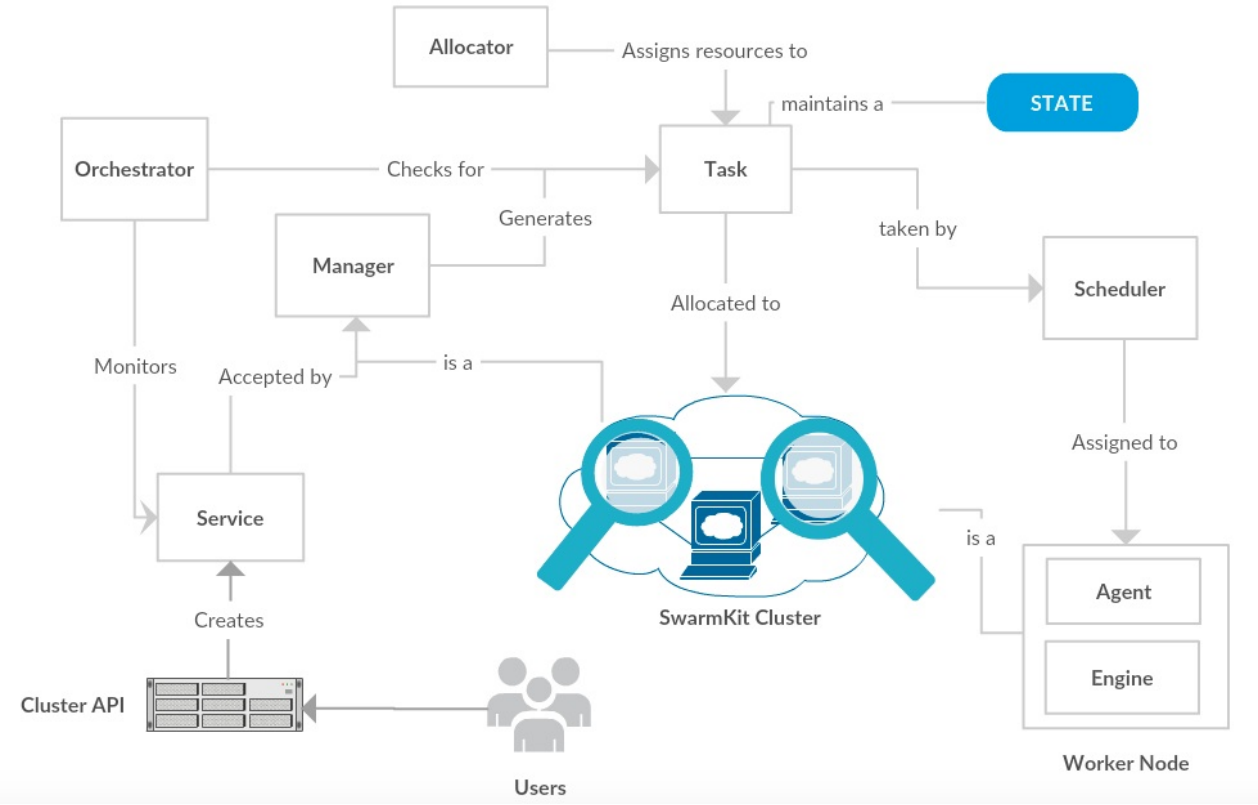
Scheduling of the services and assigning them to a node in the cluster is handled by `manager/scheduler` . The scheduler only depends on the `manager/store` , because it needs information about the different nodes to decide where to deploy the tasks.

Allocation of resources for the task execution is done in the `manager/allocator` module. This module uses the vendor library `libnetwork` for setting up an network connection with the task. `go-events` is used for the event driven communication between the manager and the allocator.

Module `manager/dispatcher` is responsible for dispatching tasks and tracking agent's health. The tracking of the agent's health is done via a `manager/heartbeat` .

The biggest module within the `manager` is the `manager/orchestrator` which runs a reconciliation loop to create and destroy tasks as necessary for global services.

Functional View



SwarmKit Functional View

The figure describes SwarmKit's important functional elements, their responsibilities, interfaces, and some primary interactions among them. It drives the shape of the overall system design and architectural decisions. It also depicts the flexibility of SwarmKit as a separate pluggable system and the ease of adding new features in any particular part of the project.

The components shown in the functional view diagram are as follows:

- **Cluster API:** It is the endpoint that is available for the users to create service requests. The valid service requests then end up as *services* which will be executed by the swarm.
- **Service:** A service is a set of instructions for the cluster about what needs to be run. It is the central structure of the cluster system and the primary root of user interaction.
- **Task:** A task basically represents a unit of work assigned to a (worker)node in the *cluster*. A task contains a description about running the container. As a task flows through the system, its **state** is updated accordingly and the change of state only flows in one direction. Once a the task is bound to a node, it can only run on that node or fail.
- **Orchestrator:** The service informs the orchestrator about how to create and manage tasks. The Orchestrator ensures that services have the appropriate set of tasks running in the *cluster* according to the *service* configuration and polices.
- **Allocator:** The allocator dispenses resources, such as volumes and networks to tasks, as per the respective task requires.

- **Cluster:** A cluster is made up of an organized set of Docker *Engines* configured in a manner to allow the dispatch of *services*. It contains two kinds of nodes:
  - **Manager** A manager accepts *services* defined by users through the *cluster API*. When a valid *service* is provided, the manager will generate tasks, allocate resources and dispatch *tasks* to an available *node*.
  - **Worker** A typical worker contains an **Agent** that coordinates the dispatch of work for a *worker* and the recipt of *tasks*. This agent maintains a connection to the *dispatcher*, waiting for the current set of tasks assigned to the node. A worker also contains an **Engine** which is a shorthand for the *Docker Engine*. It receives and executes *tasks* while reporting on their status.

Evolution Perspective

SwarmKit has evolved quite a lot since the inception of the project in Feb 2016. A lot of new features have been added to the system since then. There have been quite some contributions in terms of new features and bug fixes by the community. This section will briefly cover the evolution of SwarmKit throughout its development.

Docker was released as an open source project by dotCloud, a platform as a service company, in 2013. *Docker Swarm standalone* was then introduced as a native clustering system for Docker. This basically turns a pool of Docker hosts into a single, virtual host using an API proxy system. This was Docker's first container orchestration project that began in 2014. It is still a very convenient tool to schedule containers. Then came **SwarmKit** in early 2016. This included cluster management and orchestration features in Docker Engine 1.12 and up. When Swarmkit is enabled, the Docker Engine is said to be running in *swarm mode*. Swarm mode (SwarmKit) was Docker's response to the community's request to simplify service orchestration. Docker announced the Moby Project in *DockerCon 2017* which is an open-source project created by Docker to enable and accelerate software containerization. This project heavily influences the architectural decisions for SwarmKit going forward.

SwarmKit currently follows the semantic versioning convention when releasing an update, that is `vA.B.C` where `A` denotes a major release, `B` denotes a minor release, and `C` denotes a patch release. The project currently only has 3 releases published:

- M1 Published on 18 Mar 2016
- M2 Published on 29 Apr 2016
- v1.12.0 Published on 29 Jul 2016

The most recent release `v1.12.0` was the Tag version of SwarmKit shipped in Docker 1.12.0. There has been a lot of development on the project since then in the `master` branch and the version bump branches which follow this kind of versioning: `bump_v1.12.2`. These maintain stable versions of SwarmKit based on features developed till then for the corresponding semi-releases. The versioning in these branches is not very consistent. For example the most recent branch is `bump_17.03.1`. This is not a big issue as the next major release would go hand in hand with the next Docker release.

Besides the new features and bug fixes on each release, SwarmKit also keeps improving its transparency and product design documentation. The development of the project has been quite organized and the roadmap seems well planned to address the technical debts and service new feature requests.

Technical debt

Technical debt is a concept that represents the implied cost or additional work of choosing an easy solution now instead of using a better approach right away that is harder to implement or takes longer. This does not have to be a conscious decision, e.g. it can be a result of lack of experience or knowledge.

Identifying Technical Debt

In a big project like SwarmKit, identifying Technical Debt might feel a bit overwhelming, luckily there are different approaches to doing so.

The following approaches were used to identify technical debt:

1. Code inspection tools

2. Analyzing design and architecture
3. Issue analysis
4. Manual code inspection

One of the cases found is related to issue #2516, which will be explained in the section on design debt.

An example found by manual inspection is the `manager/allocator/network.go.` , which contains a lot of long and complex methods without comments. Another example is `manager/dispatcher/dispatcher.go` , which has 8 TODO's, however, no issues are created to actually fix them.

To realize good maintainability and understandability, it is very important to keep code documented and clean, for these cases it is safe to say that there is some form of technical debt present.

Design/Architectural Debt

This section will focus on parts of the system that contain architectural or design flaws, this means that for example, they are not as flexible as they could be. We will glance over the architectural technical debt that we identified and explain why they will require time investments to be fixed.

One of the big architectural decisions that the needs to be made is referenced by issue #2516 **Rewrite Allocator** by Drew Erny. Which states the Allocator code is badly written, quoting Drew: "is a source of constant bugs and breakages". Apparently, the code associated with the allocator has a lot of problems like:

- Cluttered code which isn't being used.
- Methods implemented straight on the Allocator object, which is suggested to be separate.
- Under-commented and tangled code, which makes piece-wise refactoring almost impossible.
- Possible inconsistencies between the raft state and the local state of the network allocator because of logic errors.
- Race conditions in local state initialization because initialization and allocation using the same code paths, resulting in IP allocation errors.

This issue was marked with the `exp/expert` tag, which means it requires hours from people with the maximum amount of experience/skills. Rewriting the entire allocator component, according to Drew, would give them a clean slate and would let them finally get rid of the most ingrained design flaws.

Identification of Testing Technical Debt

According to the CodeCov badge in the `README.md` in the SwarmKit repository, the current test coverage is 61%. This level of test coverage has been around 60% in the last 6 months, however, untested packages are not considered in the calculation.

Packages with low test coverage

The three packages with the lowest test coverage are `cmd` (the command tools to interact with the Swarm), `protobuf` (the code for the protocol buffers), and `agent` (the code for the SwarmKit agent).

Within the `cmd` package, the `swarm-rafttool` has a test coverage of 12.97%. This is caused by the low test coverage of two of the three files, `main.go` and `dump.go` have little to no test coverage. The `dump.go` file is not tested at all, the reason why is unknown, maybe due to the high cognitive complexity (191, as reported by CodeClimate) and large number of code smells (24). The low test coverage of the file `main.go` , is due to the fact that a function within a `cobra.Command` is really hard to test. A possible solution for this is by extracting the command code into another function, for example as shown below.

Before:

```
boomCmd = &cobra.Command{
    Use:   "boom <output directory>",
    Short: "Explode all the things",
    RunE: func(cmd *cobra.Command, args []string) error {
        for _, arg := range args {
            println("boom " + arg)
        }
    },
}
```

After:

```
boomCmd = &cobra.Command{
    Use:   "boom <output directory>",
    Short: "Explode all the things",
    RunE: func(cmd *cobra.Command, args []string) error {
        boom(args...)
    },
}

func boom(args []string) error {
    for _, arg := range args {
        println("boom " + arg)
    }
}
```

By refactoring the code according to the second example the function `boom` becomes better testable.

All the other commands in the `cmd` package are not tested at all. The `main.go` files can be tested using the method as described above.

Within the `protobuf` package only two files are covered by tests, the average test coverage of these two files is 28.81%. The reason for this low test coverage is that this specific code is generated.

The `agent` package has a test coverage of 49.78%. This package contains one of the two integration test in the whole project. The test coverage can easily be increased by adding tests for a number of small files, which have a low coverage. Beside that a lot of files in the package have a high cognitive complexity, due to long functions with a lot of if-statements. Splitting these functions into smaller functions makes it much easier to test the whole file, increase coverage and lower the cognitive complexity.

Testing procedures

The majority of tests are pure unit tests with a few exceptions of tests that do some sort of class integration test. There are two full integration tests: one which tests the main functions of the Swarm cluster, the other tests the agent controller flow against a docker instance to make sure it does not blow up.

The SwarmKit team wants to avoid mocking or implementing fakes for classes, because they are expensive to maintain. Due to this decision a few critical parts of the system remain untested.

```
// TODO(stevvooe): The current agent is fairly monolithic, making it hard
// to test without implementing or mocking an entire master. We'd like to
// avoid this, as these kinds of tests are expensive to maintain.
```

The SwarmKit team wants to decouple classes, which should make testing easier (as in the `cobra.Command` example) and cheaper to maintain.

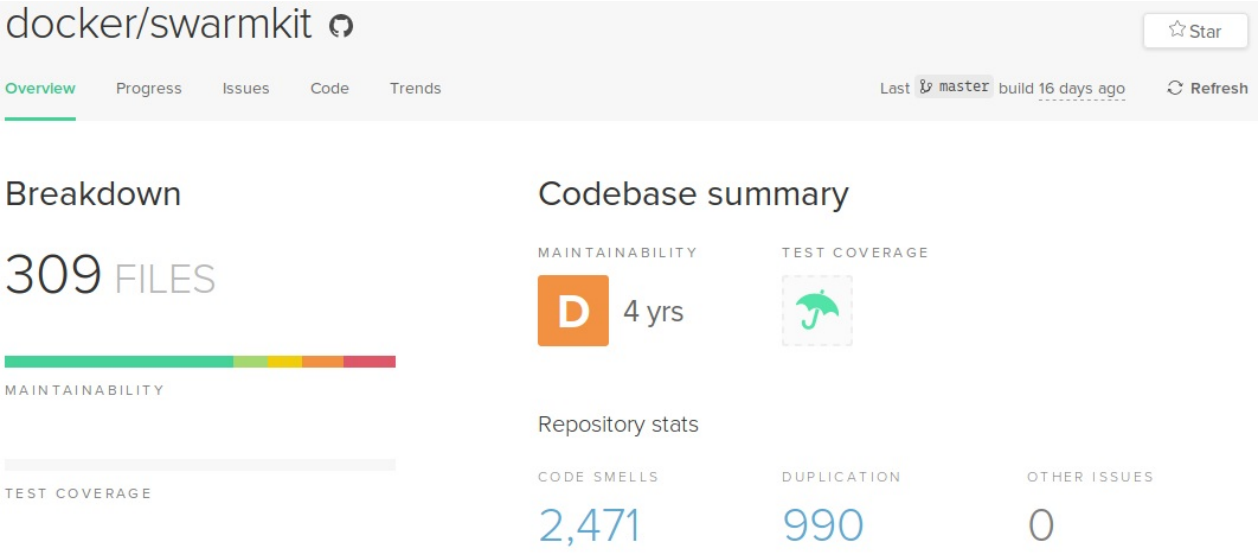
Tooling

This section describes the code inspection tools used to identify technical debt and the insights these tools provided.

CodeClimate

The SwarmKit project uses CodeClimate to assess the maintainability of the project.

At the time of writing, this is the status of the project:



The CodeClimate maintainability grade for the SwarmKit project is a D (Technical Debt ratio between 20% and 50%), with a time estimate of 4 years to resolve all these issues. Furthermore, there are 2471 counts of code smells and 990 counts of duplication marked in the project. These metrics are skewed, since a large part of the code in the project is auto-generated by ProtoBuf. A large portion of files which CodeClimate graded poorly are these auto-generated files, for which maintainability is of no concern to the SwarmKit maintainers.

Disregarding the auto-genereted files, these are files with the lowest grades and longest time estimation to resolve issues:

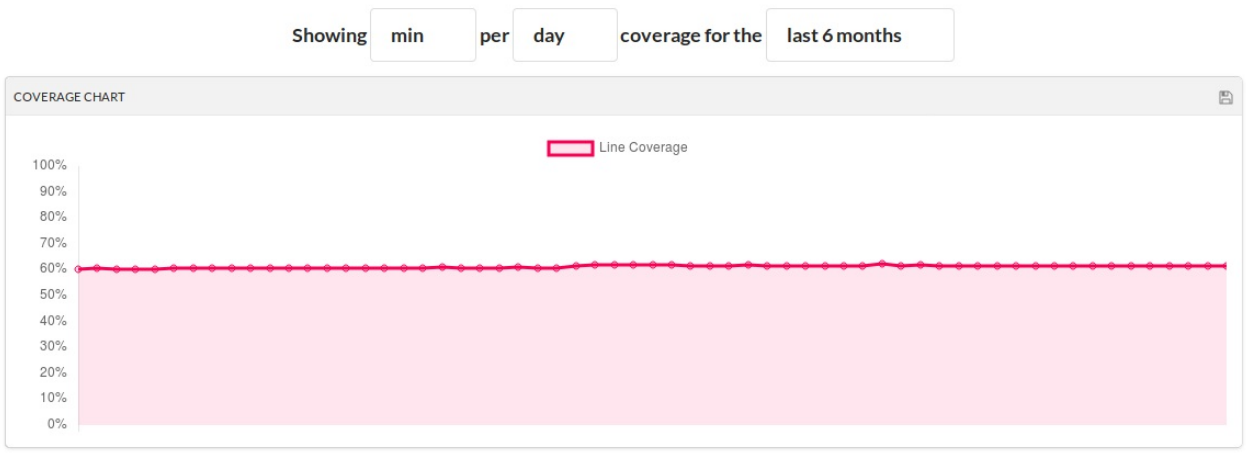
- `manager/state/raft/raft.go`
- `manager/allocator/network.go`
- `manager/dispatcher/dispatcher.go`
- `manager/controlapi/service.go`
- `cmd/swarm-rafttool/dump.go`

It becomes clear a lot of maintainability issues stem from code related to the Manager.

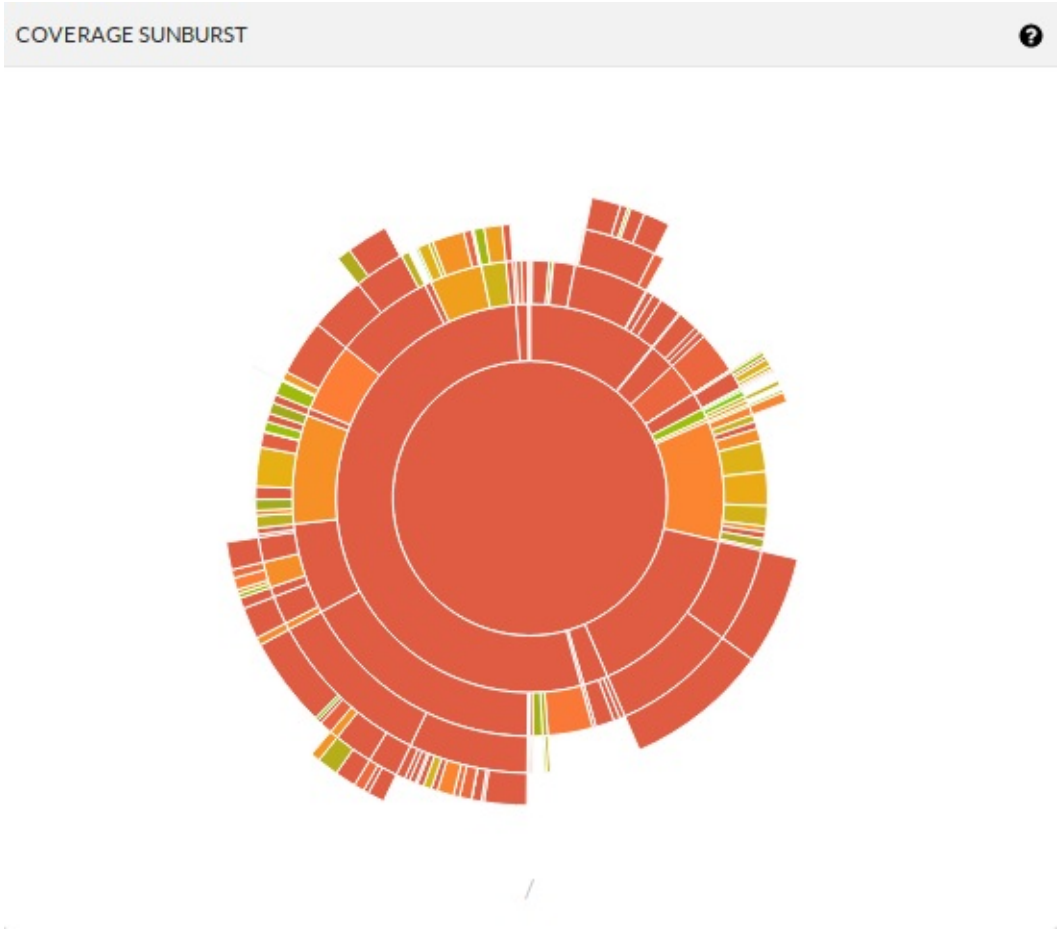
CodeCov

CodeCov is a code/test coverage reporting tool. It has a straightforward formula: `hits / (hits + misses + partials)` . Here, `hits` indicate parts of the codebase which are being executed, while `partials` are parts which are partially executed by the test suite. Finally, `misses` are parts of the code which have not been executed.

At the time of writing, this is the status of the project:



Code coverage has barely changed the past 6 months, since the SwarmKit project is actively being developed and new code gets added daily, this means that a fair amount of that code is being tested. An additional visualization is the Sunburst, depicting the folder/file structure and the test coverage of that part of the project:



This shows us specific parts of the project are well tested, while others are not. The innermost band that spans half the core is the `manager` folder, which also contains a lot of low maintainability files as described in the section on CodeClimate. This makes it easy to identify large files with low coverage, some of these include:

- `manager/state/raft/raft.go`
- `manager/allocator/network.go`
- `manager/allocator/cnmallocator/networkallocator.go`
- `manager/dispatcher/dispatcher.go`

As was to be expected, there is a lot of overlap with the files listed in the CodeClimate section.

Impact of Technical Debt

Most of the violations are Single Responsibility Principle (SRP) based, which is a result of many long files with long functions. These violations can have a big impact, because it often occurs in the core of the program. When these violations are re-factored, thus making sure a class/function has a single purpose (making them more robust), making changes will be less daunting (which is the case at the current time, as seen by the discussions between the developers).

In combination with the low amount of test the impact becomes even higher. After resolving the SRP violation there is now a way to make sure that the code does actually function the same way as before. So the emphasis is on creating a more extensive test suite, to provide a larger safety net when eventually refactoring becomes inevitable.

Conclusion

This chapter analyzed Docker's SwarmKit and it can be concluded that it is a well thought out project.

The first section throws light on the stakeholders involved in this project. Furthermore, stakeholders were analyzed through a power vs. interest grid, and concluded that the stakeholders with the highest interest and power include the core team, Docker Inc., partners and the Moby project.

The next section analyzes the dependencies of the project and visualizes this in a context diagram. It neatly separates external and internal entities of the project. After which an interview with core developer Drew Erny is discussed. By interviewing a core team member, we got a better understanding of the way the team copes with technical debt and the architectural difficulties they face.

SwarmKit has a modular nature, which makes it relatively easy to extend the project. This is achieved by common processing (for example the Common Data Store) and standardization (for example the Task Model) in the design of the project. The Development view section discusses the structure of SwarmKit's codebase. The next section covers technical debt of the project. It covers a few major architectural/design debts like the buggy allocator. The code quality and coverage is also analysed while describing techniques used.

This chapter also discusses the contributions we managed to make which helped progress the project. Conclusively, SwarmKit is a really interesting project which is professionally managed by a team of talented engineers. We loved our interaction with Drew and enjoyed working on the project. We will follow the progress of the project and try to keep making contributions.

Reference

- [1] KAEWKASI, C. and CHUENMUNEEWONG, K., 2017. Improvement of container scheduling for Docker using Ant Colony Optimization, 2017 9th International Conference on Knowledge and Smart Technology: Crunching Information of Everything, KST 2017 2017, pp. 254-259.
- [2] Mendelow, A. (1991) ‘Stakeholder Mapping’, Proceedings of the 2nd International Conference on Information Systems, Cambridge, MA (Cited in Scholes,1998).
- [3] Rozanski, N., & Woods, E. (2012). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.

Eden

Team members

Aravindakshan Ramesh	Mohammed Al-Owayyed	Myeongjung Park	Louis Sikkes
			

Abstract

Sahana Eden is a free open source system for disaster management that is highly regarded for its configurability and customizability. It provides a solution for governments, organizations, civil societies, communities and affected individuals in responding to a disaster when one occurs. The system also receives support of various types from volunteers in the community, as well as from several professional companies.

Index

Introduction

Sahana (which means “relief” in Sinhalese) started after the tragic Indian Ocean earthquake and tsunami in 2004, when members of the IT community in Sri Lanka, called the Lanka Software Foundation (LSF), began to implement a solution for efforts to alleviate human suffering during the aftermath. In 2009, LSF requested to make the software project an independent nonprofit organization based in the United States with a mission of providing reliable solutions in emergency management, humanitarian relief and social development domains. Since that time, Sahana Software Foundation (SSF) has developed many open source software products, mainly focusing on solutions used by relief organizations.

Nowadays, Sahana Eden, the latest evolution, includes many functionalities in managing organizations, people, projects, inventory and assets, as well as handling assessment information and providing situational awareness through maps. Its goal is to both plan ahead so that response can be fast and to provide a management tool during such a situation. Thus, Sahana Eden is highly appreciated by various communities, from governmental and nongovernmental organizations (NGOs), who appreciate the customizability, to the open source community and academic researchers, who see it as a useful platform for analyzing and studying open source disaster management software.

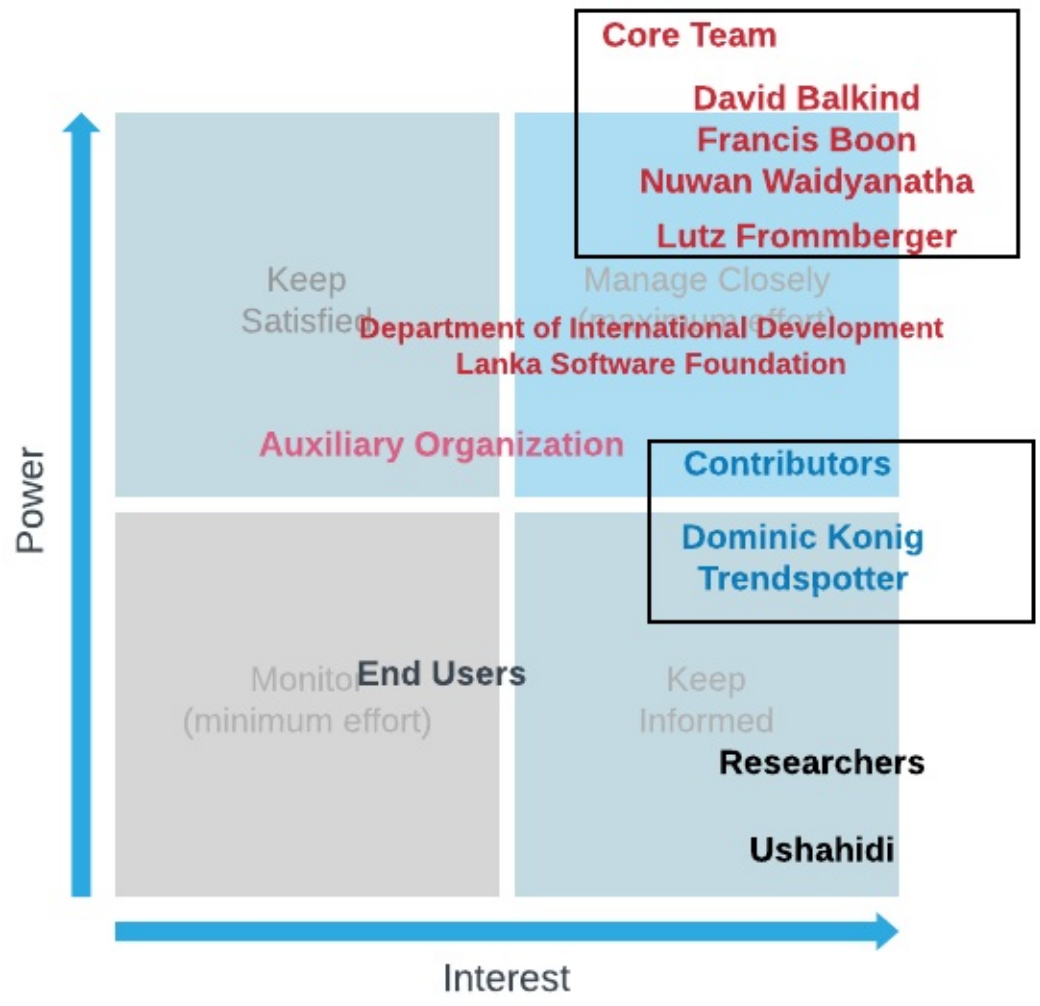
Stakeholder analysis

In this section, we identify the key stakeholders of Sahana Eden. At first, we state the board of directors of the Sahana Eden foundation, we will later refer to them as the core team. After that we list the types of stakeholders proposed in Rozanski and Woods and identify them in a table:

Type	Stakeholders	Description
Acquirers	Core Team	The core team of Sahana Eden (their board of directors) oversee the future roadmap of company by creating issues and engaging in discussions for system development.
Assessors	Core Team and Contributors	The core team assess the conformance to standards and legal regulations themselves. In this category we have identified two people: Brent Woodworth, who is an International Risk and Crisis Management expert, and John Smith, who is a law enforcer at the Disaster Risk Mitigation (DRM).
Communicators	Core Team and External Organizations	Devin Balkind, the president of the project, Nuwan Waidyanatha, also a member of the board of directors, and Lutz Frommberger post regular updates and developments of Sahana Eden on external communication platforms (their website, blog, Twitter, etc.)
Developers	Core Team and Committers	In the Sahana project the main people are Francis Boon (over 3000 commits), Nuwan Waidyanatha. They are the co-founders of the project and integral in the development of the system. They review almost all pull requests on github. Furthermore , Dominic König (over 3000 commits) and a Github user called trendspotter are also very active in the development process through Github. The pull requests and issues are handled by Dominic König and Francis Boon
Maintainer	Core Team and Contributors	The overall evolution is maintained by the core developers, all other maintenance tasks like bug fixes are done by contributors. The localization team appoints a maintainer for each language translation and the teams are governed by the core team members.
Suppliers	Core team and Auxiliary Organizations	Auxiliary organizations provide relief measures and are not directly involved with the system. Since Sahana Eden is also a self hosted system where the deployment is done by users who use the system.
Support Staff	Developers	Support for the development of Sahana Eden is done by internal as well as external developers through github and through Google Groups
System Administrators	Core Team and Users	Since Sahana Eden is self deployed, the user becomes the system administrator
Testers	Core Team and Contributors	The core team is involved in fixing bugs and reporting errors. In addition, Sahana community members can sign up as testers to participate in the testing of development branches. trendspotter is one of the main testers of the system.
Users	End Users	The end users represent most of the Sahana Eden Community. They range from volunteers,health care organization or government organizations. They are mostly concerned about the functionality of the system.

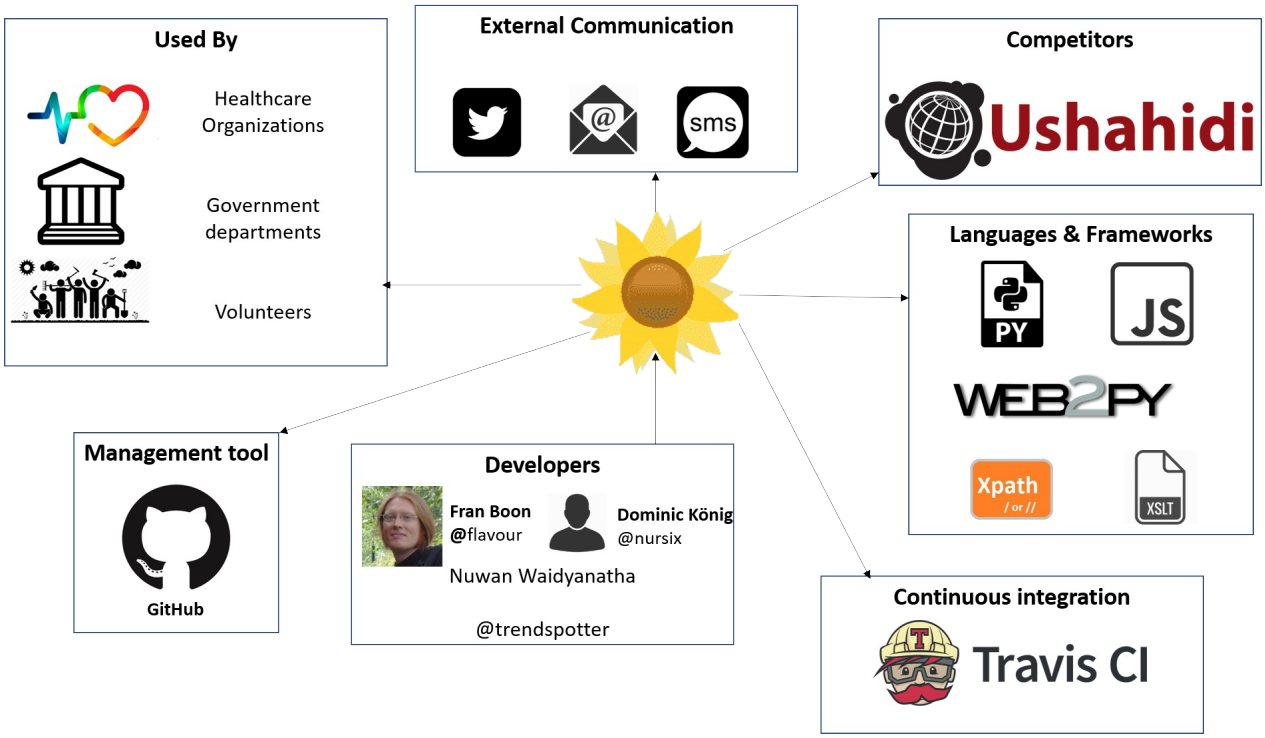
Going beyond the book we have identified stakeholders which do not directly fall under the Rozanski and Woods classification.

- Competitors: Ushahidi is an open source disaster response organization that allows anyone to gather distributed data via SMS, email or web and visualize it on a map or timeline. Their goal is to create the simplest way of aggregating information from the public for use in crisis response.
- Researchers and Scientist: They research and present findings of the system through journals. They are also integral in improving the functionality of the system. For instance Pictographs in Disaster Information Communication for the Linguistically Challenged is a core-recognition project, funded by the Humanitarian Innovation Fund of UK.
- Media: They present unambiguous and reliable sources of information and is vital in the communication of the system to people and developers who are interested in new challenges and projects. Sahana software and its role supporting post-tsunami disaster relief in Sri Lanka were featured in the 2006 BBC World documentary The Code-Breakers.



Power interest diagram: We have analyzed the stakeholders and grouped them in a diagram to give a better understanding of who to monitor and how.

Context View



The context view describes the relationship, dependencies and relationships of the system with its environment. In the diagram above we have split these into groups that we have derived from the stakeholder analysis and the external entities that we will now discuss.

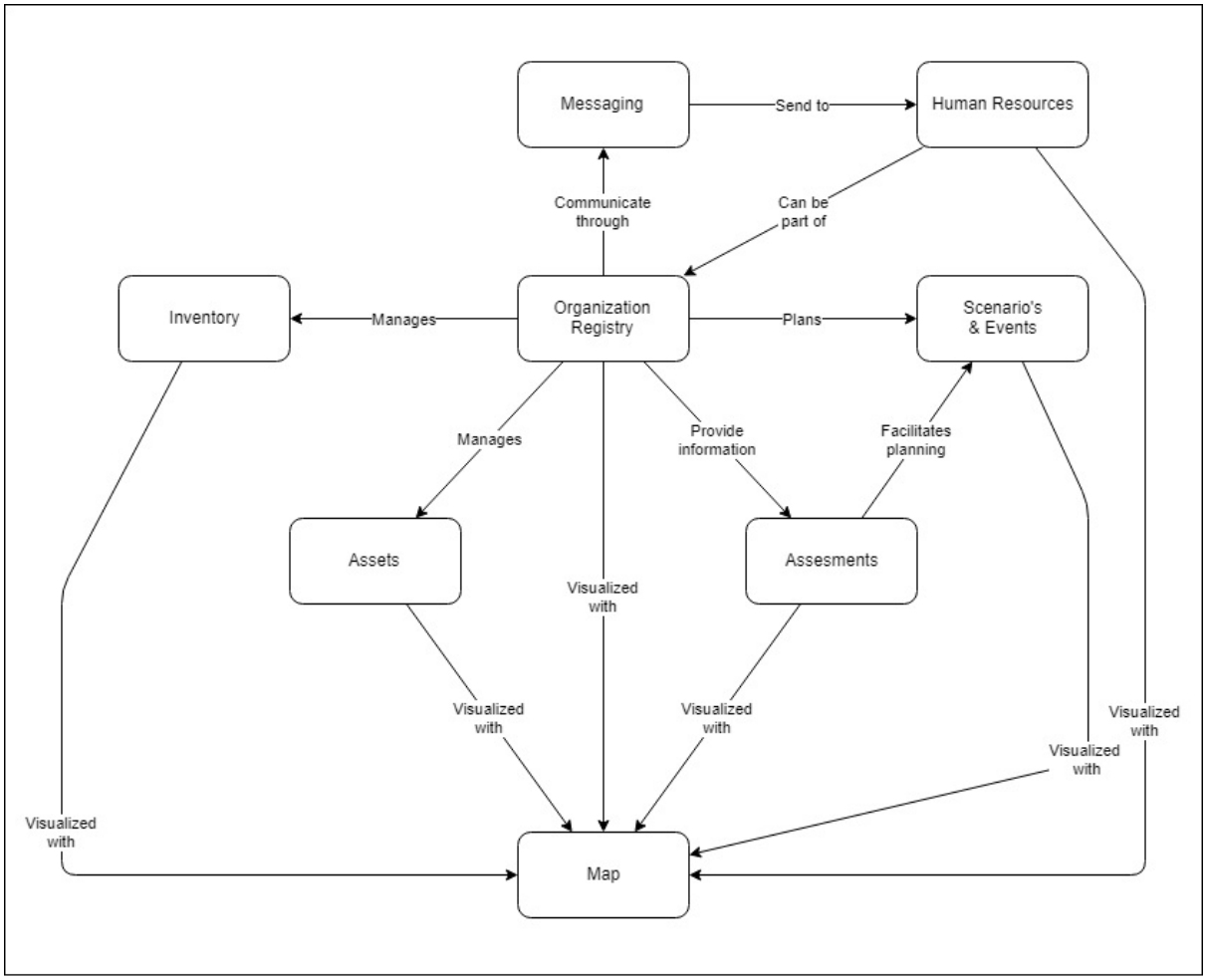
- Governmental and non-governmental organizations: The external entities contained in this group range from government departments and the police to healthcare organizations like the Red Cross. These organizations make use of all external interfaces of the system. We will describe the three most important ones now: alerting, importing and exporting and printing maps. These will be explained later in the document.
- Volunteers: Besides big organizations, individual volunteers can also help during a disaster. These people will not have a wide variety of assets or inventory and will therefore not use all external interfaces of the system. The main focus of these people lies in receiving alerts whenever something happens or letting organisations know their skills and availabilities.
- Media: The last external entity that we discuss is the media. They are not involved in participating to support the disaster area, their main focus is to report on the activities. They will therefore only use the alert functionality to be alerted of any activities.

Development View

Before we start to contribute to the Sahana Eden, we have to know how the system is built. In this document, we start off by giving an overview of the components of the system. In the second section, we discuss the common design models and approaches that were used to build the system. Thirdly, we talk about the guidelines that are being used during the development of the system, we do this by talking about the release and testing processes.

Component overview

Sahana Eden contains many different modules which can be configured to provide a wide range of functionality. Since these modules are easily customizable, they can provide solutions in a wide variety of contexts. This further enhances by the fact that these modules can be enabled or disabled to provide the needs of the deployer. In this section, we will outline these and explain what purpose they serve. The information found here has been extracted from a brochure found on the Sahana Eden website [3].



Module Overview: After analyzing all modules we have created the following diagram to visualize their dependencies on each other.

This diagram gives a logical view of how the modules work together, we have not extracted this from the code. We can see that in the middle the organization registry interacts with most other modules. We then see that organizations can perform a variety of tasks like managing inventory or assets. Finally, most modules are linked to the map, which can create a map of most data in the system.

We have identified the following modules:

- Organization Registry: This module allows registry, searching and modification of data of organizations. It also allows the registration of warehouses, offices and field sites which can be mapped to other modules.
- Project Tracking: The project tracker provides a platform to allow the organizations to manage the projects that are currently active and what their needs are. This tool is used for instance by the Disaster Risk Reduction Project Portal [13].
- Messaging: The project allows users to set up groups so a large audience can be messaged at once. Furthermore, there is the possibility to subscribe to communication channels such as email, SMS, Twitter and Google Talk.
- Scenarios & Events: This tool can be used to map the needs in different scenario's, including human resources, facilities, assets and tasks. There is the option to create specific templates which can be used whenever a new disaster happens.
- Human Resources: Eden is able to track volunteers and staff of different organizations. It keeps track of what they do, where they are and what their skills are.
- Inventory: A wide variety of items, ranging from supplies for survival to tools for rebuilding area's, are often needed after a disaster. This module can keep track of inventories and match requests. It can be used to record and automate transactions for sending and receiving shipments.
- Assets: Vehicles, radio equipment, power generators and more are all needed after a disaster. This module provides a platform to track which assets are available, where they are and in which condition they are.
- Assessments: It is used to collect and analyze assessment information to support planning ahead. Users can design templates that can later be imported when a disaster happens. The module includes reports, graphs and maps.
- Map: The integrated map module can be used to visualize data. It supports any location-based data and aims to provide situational

awareness. Many formats for overlaying data are supported such as population and weather data.

Common design models

The Eden project is built in two languages, Python and Javascript. They use Web2Py with the goal of making the web development easier, faster and more secure. This framework enforces to build an MVC environment. The other big framework that is used is the S3 API. They use this for a lot of their functionalities: authentication, authorization and accounting, logging, as a RESTful API, accessing the database, exporting/importing, GUI's and mapping. Finally, to import and export data they use the XSLT format. This allows them to import and export data easily and even on-the-fly. Their repository contains 2 folders dedicated to defining the resulting structure of the XML files. Furthermore, the import/export templates are stored in the static/formats/... folder. We will now delve more specific into two design models: internationalization and importing and exporting.

Internationalization

Since different people from different countries and cultures have to interact with this system it is important that the system is understandable for all of the users. Most of the languages are handled using the web2py localization engine. This contains a file for each language which contains the translations from English to the correct language. Some of these files have been established using Google Translate.

Importing and exporting

Since Sahana Eden is allowing organizations to collaborate each other it has to import data from most of these organizations. This data can range from inventory information to data about a certain area. To support a wide variety of different formats to use for importing and exporting data they use XSLT templates. This can be used to create for instance csv and XML files on the fly.

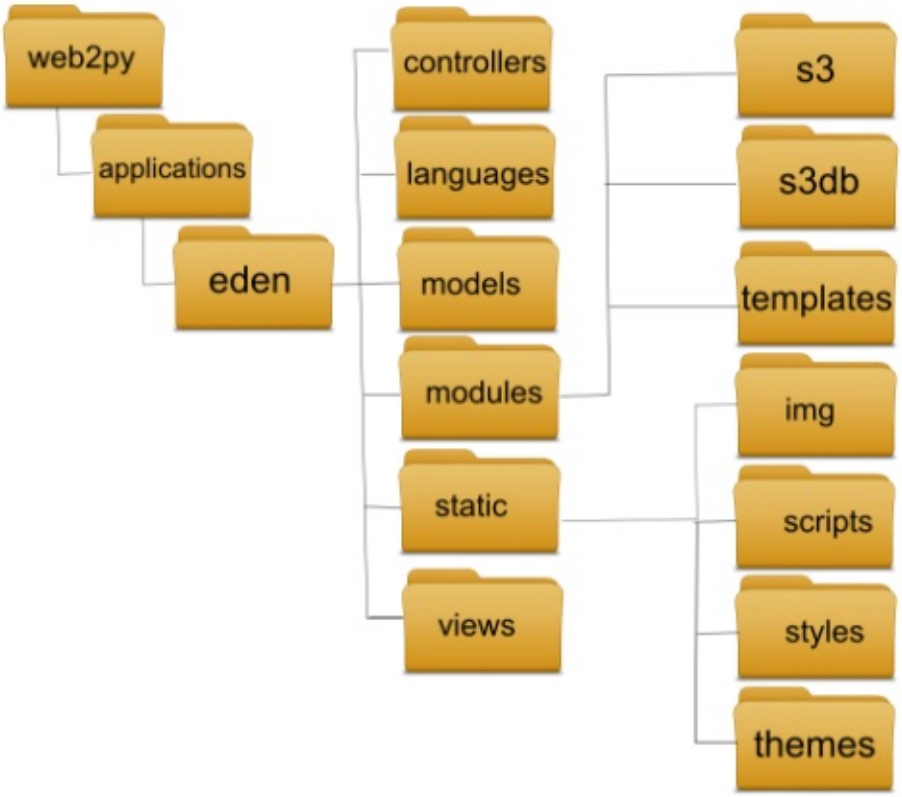
Code conventions

A full list can be found here [15]. To check for these code conventions they use several tools including PEP8, PyLint and PyChecker. Since they aim to be internationally available, all strings should be labeled, so that the appropriate language can be gathered from its corresponding file. Furthermore, all files, classes and functions, should have docstring to allow auto-generation of API documentation using Epydoc.

Codeline model

In this section, we will describe the components that make up the code of the project. For this, we have studied their Github repository [10] and analyzed the components that can be seen in the picture below. This picture was found on their website. We expected to find some of the module structure that we have found earlier in this document to be present in the code. This is not the case, all the functionalities from these modules are handled in the controller folder, which has no further structure. We will discuss this later on.

We can see in the picture below that there are 6 main folders. In the repository itself, there are four more folders on the same level. These folders are 'cron', 'docs', 'private' and 'tests'. In the 'docs' and 'private' folders, we found some documents unrelated to the rest of the code so we will only not analyze these any further. While observing the repository we found very minimal use of a folder structure within these specified folders, most of them contain a bunch of files that do a wide variety of things.



Folder structure of the repository

- Model, View & Controllers : The controllers are the part of this system where the logic happens (.py files). Instead of a further structure inside this folder, we found some single files that are related to a module. For instance, the 'project.py' file handles project registry and the 'inv.py' file handles managing the inventory. The models are .py files and the views are HTML files, both folders have no further structure.
- Languages : The files in here are Python files that translate English to another language. The format of the files is as follows:
  {
  "%s rows deleted',nrows": "%s lignes supprimées',nrows",
  ...
  }
- Modules : Contains external modules like GeoJSON (an open standard format designed for representing simple geographical features), S3 (accommodates cloud object storage), S3 database, S3 unit tests, ClimateDataPortal, GeoPy, a name parser and PyGSM.
- Static : The static folder contains any static files that are stored once and rarely touched afterwards. In here we can find images, themes, styles, fonts, scripts and more.
- Tests : Contains the tests, we will talk about these later.
- CRON : To perform some time-based jobs they make use of CRON. In here we find only 3 files of which one is empty and 1 contains only 1 line that sets a heartbeat. The final file is an SMS handler that probably sends automated messages.

The release process

Whenever new contributions are ready to be published they will plan to make a release version of the system. This consists of:

- Creating a QA branch in which final integration testing will be done before pushing it to the Stable branch
- Creating a Stable branch, deployment will be done from this branch
- Adding appropriate tags to the release, the following schema will be used [Branchname]-[Major].[Minor].[Sub]
- Building upgrade scripts, the pull script has been modified to call this script automatically

The testing process

An important step in the development process is verifying that the system works properly. To this extend Eden has implemented several testing approaches that we will discuss in this section. In the technical debt, we have tried to run these tests. We will discuss them further in that chapter.

EdenTest

EdenTest is a Robot Framework based test framework used for automated testing in Sahana Eden. The files implementing these tests are .txt files that contain almost regular language instructions, for instance: "Open Advanced Filter Options". These functions are defined in separated files which makes it easy to reuse them.

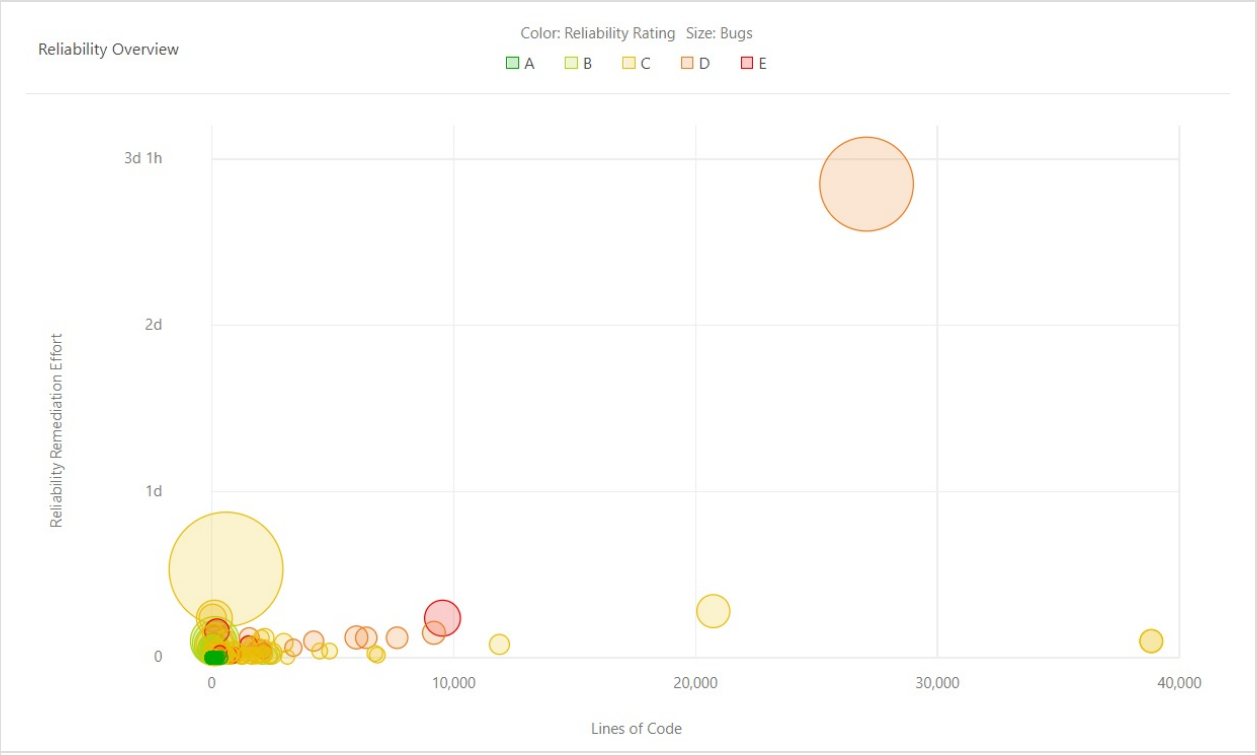
Unit Tests

From the website, we found that unit tests are used to detect problems early during the development process. This indicates that they make use of test-driven development. Each module has their own test set that was separately run. Aside from validation they use these tests to validate the design against requirements, keep implementation simple and focused and mention that this can be a great source of code samples as to how to apply your API methods. A Continuous Integration server that uses Travis has been set up. Whenever a pull request is created all unit tests are run before and have to succeed before it can be merged.

Technical Debt

Bugs & Reliability

SonarQube rated Sahana Eden's reliability as an E (from A to E), which means there is at least 1 blocker bug presented in the program. Upon further inspection, it was discovered that SonarQube flags Python 2 code as errors if the code is not compatible with Python 3. Thus, Python 2 -specified errors were eliminated to judge the code quality fairly. Bugs that do not create errors or affect the program’s behavior are still present in the major and minor code, but with an overall better rating of B instead of E. Most of the major issues were due to the absence of HTML5 format that SonarQube checks (e.g., deprecated elements such as center), similar to the Python 2 issue discussed previously. Overall, the code is well-written and reliable with few to no bugs affecting the behavior, but could benefit from using more modern versions to keep all files up to date (i.e., Python 3 and HTML5).

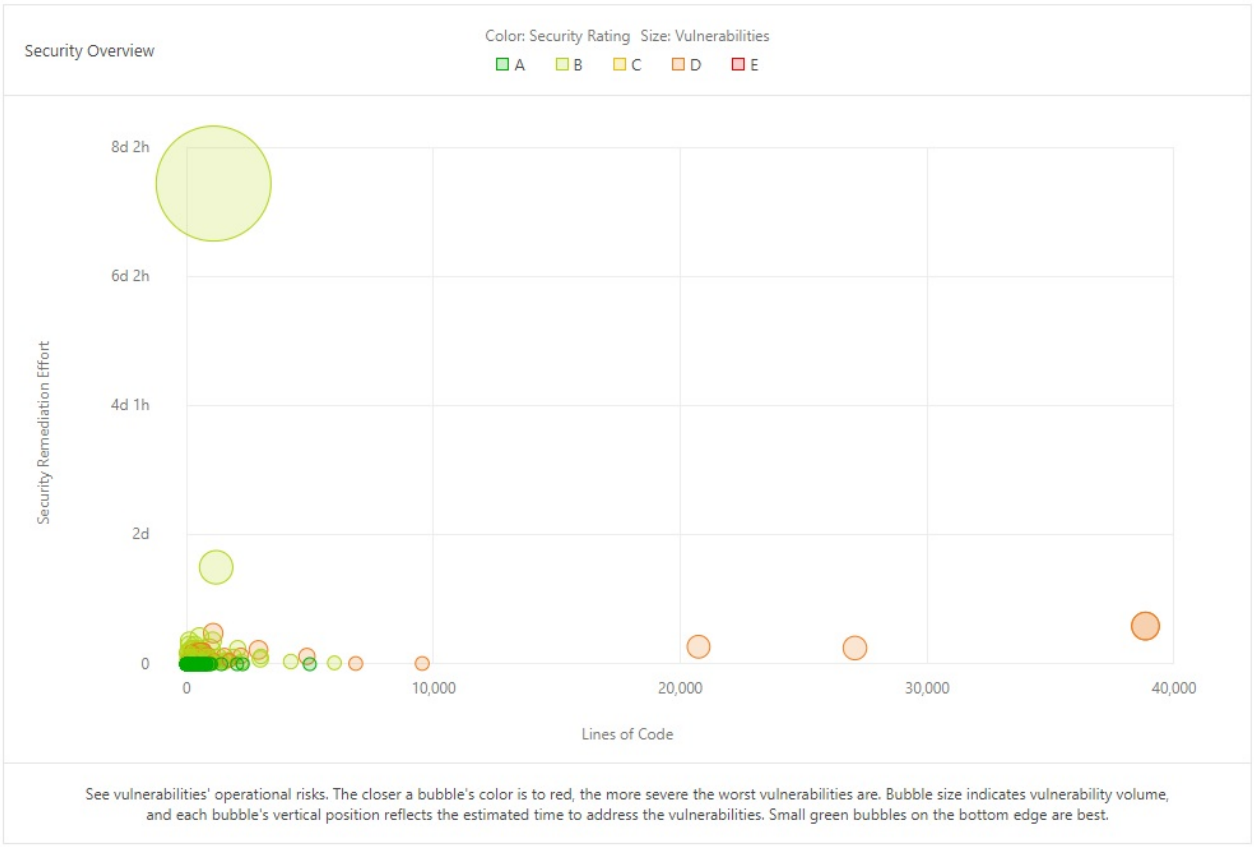


An Overview of reliability and bugs

Vulnerabilities

In this section, we will discuss the vulnerabilities that were found by SonarQube. It found a total of 434 vulnerabilities and gave the rating D since there was at least 1 critical vulnerability. 97 of the vulnerabilities were marked as critical and 337 as minor. In the graph, we can see the overall severity of the vulnerabilities and how long it would take to fix them. We notice that there are some small volume severe vulnerabilities that should not take too long to fix, but most of the code is secure.

We first analyse the critical vulnerabilities, which only 1 type of issues was found: validating arguments before making a function call. Although this is faster than dynamically evaluating the code, it can expose the program to random, unintended code which can pose a security risk. When analyzing the minor vulnerabilities are concerned with hardcoded IP addresses (like localhost) or debug code still there.



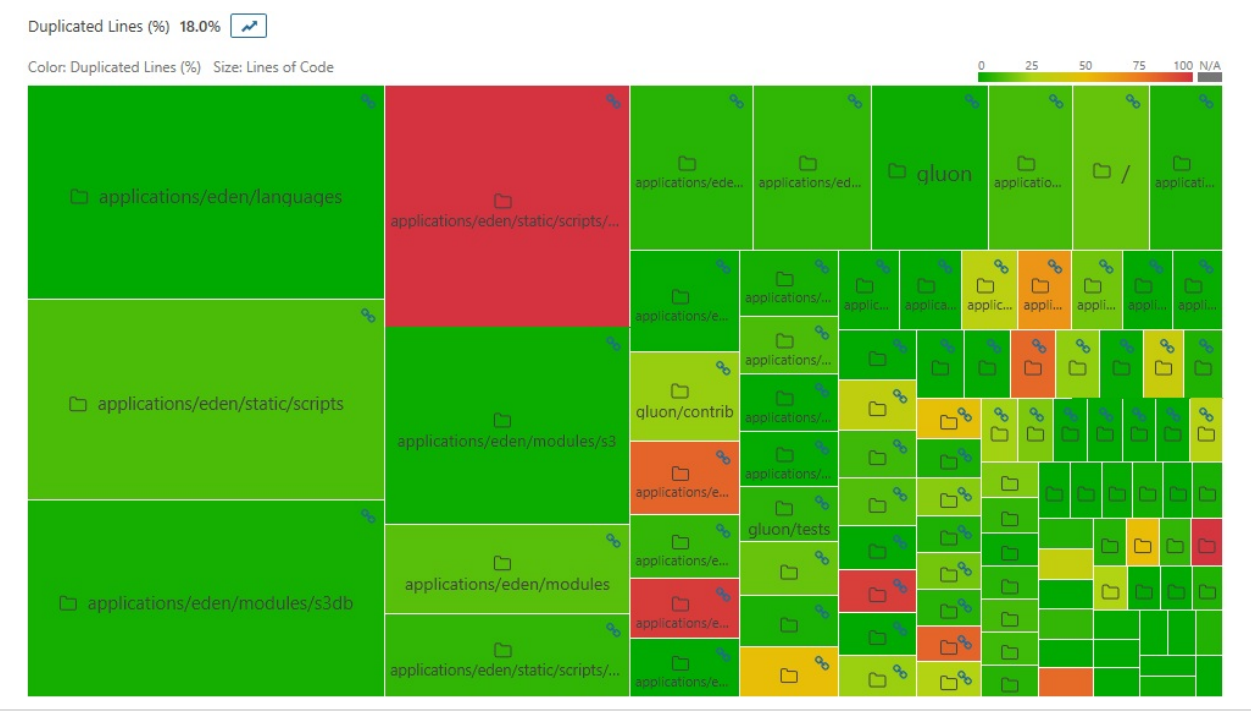
Code Smells

In this section, we talk about the Code Smells which we derived from SonarQube. We basically look at how simple or how complicated the code is and look to enhance their code with solutions which we deem can satisfy the technical debt and improve the overall quality of the code. SonarQube found 14.000 code smells. The minor errors mostly deal with renaming local variables and removing empty statements, due to time constraints we leave it out not because they are not important but they do not need immediate attention right away. SonarQube gives a maintainability rating as A since the technical debt ratio is less than 5%.

The below comments are treated as critical and major by SonarQube. These errors seem to be repetitive. One example is an initialisation function that has 16 parameters, this is too much. Another example is an assignment to a variable that is not used later on.

Duplications

We have found several files that were completely duplicated. Below in the figure is a Treemap, generated by SonarQube of duplications on directory level in the Sahana Eden project.



Treemap of duplications

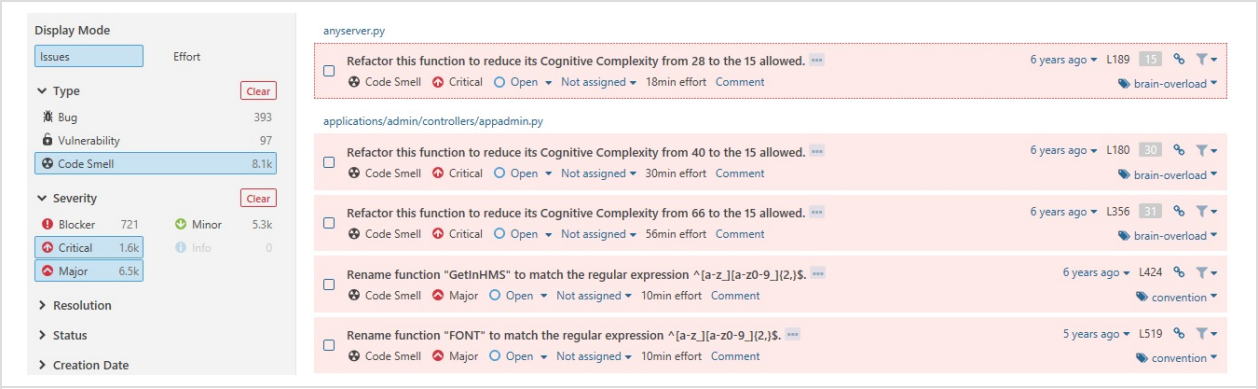
The color of the squares indicates the level in which duplicated code exists. Red indicates a high percentage and green indicates a low percentage of duplications. There are not a lot of duplications when looking on the folder level, some functions are duplicated. For example, mouse move, touch move functions, calculation functions, etc. However, if we move into class level, there are a lot of duplicated files, even in the directories that are green in the Treemap.

The density of the duplicated lines is 18%, which we find quite high. Most of the files that are duplicated are HTML or JavaScript files. These mostly occurred in modules, especially the templates and static/script folder. The files that are duplicated are often functions such as list filter, update, appadmin, config, cache or auxiliary. The percentage of duplications is quite diverse throughout the files. There are files which are 100% duplicated and some files even have the same name with a different location. Some duplicated files are in the same location with a different name. This is in violation with one of the SOLID principles: the single responsibility principle. Most highly duplicate files are about display, comments, list, and layout. Although these duplicates are not critical issues yet, these features can become problems later on and should be resolved as soon as possible. Removing these duplications would play a key role in reducing the technical debt.

Historical Analysis

In this section, we identify how Sahana Eden has attempted to improve the technical debt over the years. We will discuss how they found and solved these issues. The tools that we will use to analyze this are SonarQube and Github statistics.

First, we filtered the major and critical issues with code smells in SonarQube. The result is the following:



Historical analysis by SonarQube

As can be seen in image, most files have not changed since they were created 6 years ago. This image only shows the code smell part. Also, bug and vulnerability sections showed almost the same results. One thing we could find latest changes in is vulnerability. The changes were in simple syntax errors (e.g., if (delay == 'undefined') to if (delay == undefined)). However these changes are not the issues detected by SonarQube.

Code frequency in Github

The image above shows the code frequency by means of the additions and deletions per week. The significant changes were 6 years ago when they were created. After that, the code frequency did not show any significant changes over the years.

Secondly, Eden project core members and developer are clearly aware of the technical debt and bugs. There are 'bug' and 'enhancement' labels and they are also divided by 'Major' and 'Minor' issues that are well-explained. The developers identified the problems and how (not) to fix them, as can be seen in PR#1348. In this example, they are aware what the problem is, and refactored the code to solve them. This PR has been merged, some of the technical debts have been solved as refactoring as well.

Testing

In the development view, we have mentioned that Sahana Eden uses a couple of different tests. These tests were executed and we will report the results in this section.

EdenTest

The EdenTest is a Robot Framework used for automated testing. In order to install it, we had to download a few packages including Selenium. We followed the guide for running the tests but found that not all steps that had to be taken were explained. There is a section that explains how to create a configuration file but it does not explain how or what we should change in here. We also had to erase and populate the database (using a script in the project) before running the tests but the database stayed empty while running this. As a result, a lot of lookup functions failed since the entry that they were searching for did indeed not exist. We found that most basic functions were tested, but no attempts to break the system were made in these tests.

Unit Tests

On their website, they mention that they use unit tests to detect problems early during development. This indicates that they use test-driven development. We observe a total of 610 tests which we ran from the command line. All tests succeed so we inspected what was tested in these tests. Most functionalities are tested like parsing, initialising and other functions, with again only 1 test per function / functionality. The documentation did not describe how to generate code coverage.

Other Tests

Here we will discuss the other tests that were incorporated. Smoke Tests simply click on every link within the Sahane Eden project to check for broken links or errors made. They ran successfully and reported nothing was broken. These tests are good to make sure that you have not accidentally overlooked something. They do however not give any guarantee that what is shown is what should be shown. When running the script for the Role Tests these did not seem to be working. The documentation mentioned that these are limited at the moment. Finally, we ran the Benchmark Tests which gave us some numbers but no further explanation was given about these numbers and although they mention that the result of these tests will rate the performance relative to the system that we ran it on, we could not find such a result.

To conclude the tests and their documentation have a lot of room for improvement. We found setting up the tests to be harder than it should be because of a lack of documentation. There was also no mention of code coverage or how to set it up. We saw that most of the functionalities were tested, which is good. More tests could be implemented to check for corner cases to try and identify bugs or vulnerabilities in order to increase the solidity of the system.

Internationalization Perspective

The rationale behind choosing this is that Sahana Eden operates in various languages for different regions as per the needs of their deployment. As a disaster management system, including different languages would help to deploy the system whenever a disaster occurs, as the system must be understandable to the interested parties. This perspective was chosen because the Sahana foundation should appeal to international society, and to achieve that, it is not enough to only translate the right meaning of words; the right orientation (some languages are written from right to left), character sets, currencies or any other localization issues must be addressed, as well.

As stated before, Sahana Eden was deployed following the 2010 Haiti earthquake. Since then, more than 60 different nonprofit or government agencies used it as means for disaster management [1]. The system has been deployed notably for wildfires in Chile (2012), earthquake and tsunami in Japan (2011) and Flooding in Colombia (2011). Various organizations adopted Sahana Eden, such as Asian Disaster Preparedness Center (ADPC) for its Disaster Risk Reduction Projects Portal, International Federation of Red Cross (IFRC) and National Disaster Relief Services Center (NDRSC).

Sahana Eden has 39 different languages files [2], which mostly include translations for the majority of the text shown on the website. The software inherits Web2Py localization architecture, which simplifies the process of adding a new language. Also, files could be edited directly from the GitHub repository. Furthermore, a tool is recommended in Sahana wiki for auto-translating using the Google Translate API, which explains the literal translation that we found and fixed in the Korean language file. However, languages that are most likely to be used due to their associations with targeted areas (e.g., conflict zones and disaster areas) have an understandable but incomplete translation. The small percentage of untranslated words are presented in English, which is the default language for the system. Regarding the orientations for right-to-left written languages, those languages have an inversed web page layout to correspond with its flow. As can be observed from Arabic language page that displays the menu on the right instead. Furthermore, all characters from different languages are shown correctly, as it uses UTF-8 for encoding. Another inspected issue is currency and unit converting

between different regions, which was avoided by letting the user who inputs the amount has to enter the currency or the measurement unit as well, with the exception of capacity that always follows the metric system (m^3). The date format used is fixed across different languages (dd-mm-yyyy).

Deployment View

Sahana Eden is highly configurable so that it can be used in a wide variety of different contexts and easy to modify in order to build custom solutions. Different levels of support are available from both the voluntary community and professional companies. Especially, Sahana Eden can be accessed from the web or locally from a flash drive, allowing it to be used in environments with poor Internet. Local & web versions can be configured to synchronize to allow data to be shared between them.

The deployment view looks at parts of the system that are relevant one the system has been built or deployed. It defines computational, physical and software-based requirements for running the system. First of all, Sahana Eden runs on Python and JavaScript and hence requires Python 2.7 and JavaScript 1.8 respectively. Sahana Eden can be downloaded from the Sahana Software Foundation and installed with its requirements. In order to contribute to code, the developer should have your own repository on GitHub, a community collaboration platform based on the Git distributed version control system. The other dependencies are as follows :

- Web Server: Apache is preferred but other web servers such as Cherokee can also be used.
- Operating System: For production installations, Debian Linux v7 "Wheezy" is recommended as this is the environment for which the most support is available. Windows and Mac OS X are possible, but only recommended for single-user environments.

Hardware Requirements:

- A virtual server executing the main functional elements of the system and allowing users to access the system should, at the very least, have 1GB of RAM and 4GB HDD.
- Sahana Eden must be installed on a medium in which Python programming language can be run and where any database systems such as PostgreSQL database is supported.

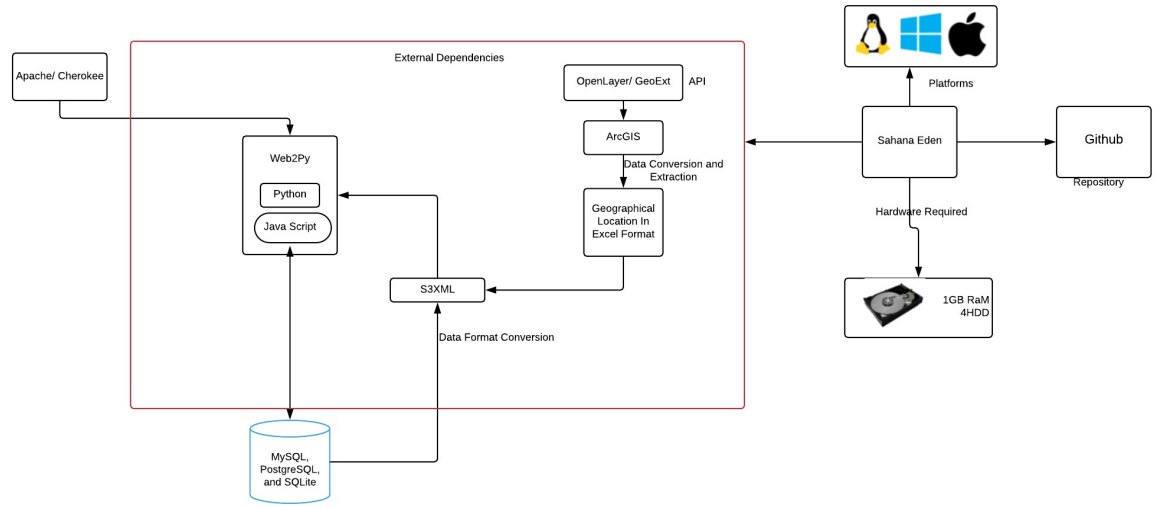
Database Settings:

- It is recommended that production systems use PostgreSQL or MySQL rather than the default SQLite. For these databases, it is more secure to provide the application with a database account with minimal privileges.
- S3XML is a data exchange format for Sahana Eden.
- S3XML is a meta-format and does not specify any particular data elements. The interface is entirely introspective to the underlying data model, thus the specific constraints defined in the data model also apply for S3XML documents.

Web Application Framework: The database is examined using Web2Py's interface.

Mapping API:

- API allows developers to be able to display customised data output relevant to a specific module on the Map.
- Sahana Eden's mapping client is based on OpenLayers & GeoExt.
- OpenLayers provides access to a wide range of data sources, from public services based on OGC standards.
- GeoExt provides UI widgets to allow the user to interface with the map.



Deployment View of Sahana Eden

Conclusion

Sahana Eden has a solid foundation with a passionate set of developers and contributors. Ever since it's initial release it has accomplished a lot and is still evolving constantly adding new modules and technologies to its platform. However, we have identified rooms for improvement throughout this document which is summarized below.

Sahana Eden consists of different modules, used for managing disasters and planning ahead. These modules can be enabled or disabled to create custom solutions in different contexts. In the code itself, we have not found this module structure but only a basic structure. The project incorporates different testing methods to guarantee all parts of the system are working. Overall we found that the mapping of the codebase is rather poor and we see room for improvement here.

Sahana Eden has a varied analysis on technical debt. Using SonarQube they were rated B for Bugs, D for vulnerability and A for Code Smells. On analysing duplication of the code we realized that the density of duplication is too high. Although this is not directly harmful, this is still a potential weakness that lowers the quality of the code. Vulnerabilities were examined and certain functions can expose the program to random, unintended code which can pose an operational and security risk.

Even though, Sahana Eden could improve from a developer's view, their main focus is to help people in need which they are doing a commendable job.

Resources

[1] Sahana Eden deployments - https://sahanafoundation.org/eden/deployments/
[2] Github language files - https://github.com/sahana/eden/tree/master/languages
[3] The website of the Sahana foundation. It contains information about who is contributing, guidelines, a blog and also links to other useful resources. - https://sahanafoundation.org/ -
[4] A book describing the system - http://archive.flossmanuals.net/_booki/sahana-eden/sahana-eden.pdf
[5] The demo of the system was used to get a good overview of the system. - http://demo.sahanafoundation.org/eden/
[6] Google Groups for discussions - https://groups.google.com/forum/#!forum/sahana-eden
[7] Twitter - https://twitter.com/sahanafoss
[8] Facebook - https://www.facebook.com/SahanaFOSS/
[9] XSLT Templates - http://eden.sahanafoundation.org/wiki/XsltTemplates -
[10] Sahana Eden Github repository - https://github.com/sahana/eden
[12] Sahana Eden brochure - https://www.slideshare.net/SahanaFOSS/sahana-eden-brochure-10577413
[13] Disaster Risk Reduction Project Portal website - http://www.drrprojects.net/drrp/

[14] Sahana Eden Wiki - http://eden.sahanafoundation.org/wiki/
[15] Sahana Eden coding conventions - http://eden.sahanafoundation.org/wiki/DeveloperGuidelines/CodeConventions
[16] SonarQube - https://www.sonarqube.org/

Elasticsearch - The Heart of the Elastic Stack

By Mathias Meuleman, Bart van Oort, Menno Oudshoorn, Mark van de Ruit

Delft University of Technology, 2018

Elasticsearch is a distributed, RESTful search and analytics engine. It lies at the heart of the Elastic Stack: a group of multiple applications developed and managed by the Elastic company. The Elastic Stack provides a way to reliably and securely take data from any source in any format, and search, analyze, and visualize it in real time.

We, four master students of the TU Delft, have analyzed the architecture of Elasticsearch and aim to provide insight into the system from different viewpoints. We do so by first identifying the stakeholders of Elasticsearch, after which we put Elasticsearch into context. We then look into the module organization of the system. Furthermore, we analyze how information is handled within Elasticsearch and how performance and scalability are monitored and upheld. Finally, we dive into the system to identify technical debt and propose ways to decrease it.

Table of Contents

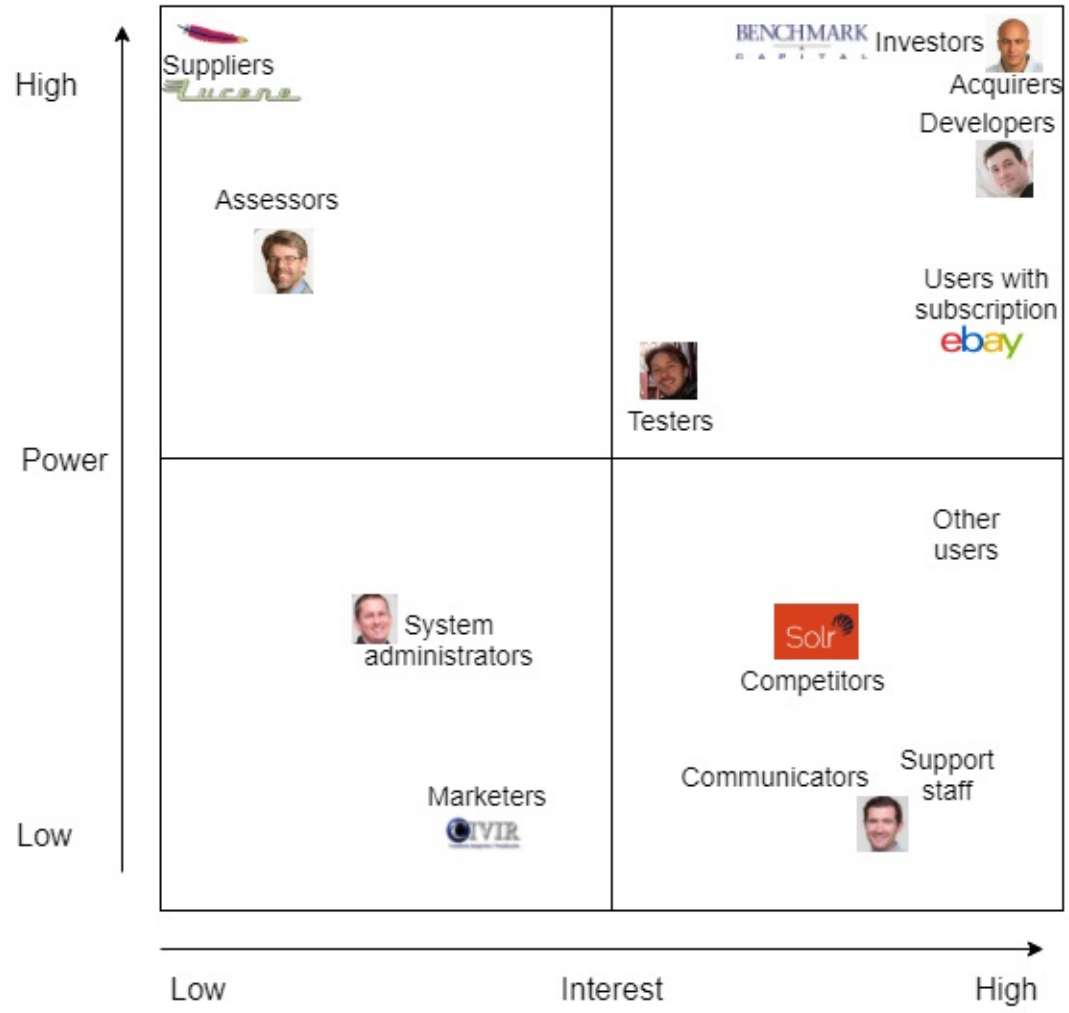
1. Stakeholders

The stakeholders of Elasticsearch are categorized according to the stakeholder types described in Rozanski & Woods (2012), including three additional types: competitors, investors, and marketers. The categories have been placed in alphabetical order.

Type	Stakeholders
Acquirers	The Elastic company itself, mainly its founders, leadership, and board.
Assessors	Baird Garett (Senior Vice President of Legal) and Robin Sharpe (Vice President of Operations).
Communicators	Marty Messer (Vice President of Customer Care) is the main responsible person for the training and consulting of the customers of Elastic. There is also a three-day Elastic{ON} event, as well as a number of paid online training courses and private courses on location.
Competitors	Apache Solr is the most direct competitor, as it also uses Apache Lucene. Other competitors include Sphinx, Hawksearch, Commvault, and SwifType.
Developers	Every contributor on GitHub. The three biggest contributors we identified are Shay Banon (@kimchy, creator of Elasticsearch), Simon Willnauer (@s1monw, founder) and Jason Tedor (@jasontedor, currently most active developer).
Investors	Elastic company. Elastic is funded by three main investors: Benchmark Capital, Index Ventures, and New Enterprise Associates Inc. (NEA).
Integrators	Members of the Github Elastic organization. The most important integrators are Jason Tedor, Luca Cavanna (@javanna), Christoph Büscher (@cbuescher), Boaz Leskes (@bleskes) and Colin Goodheart-Smithe (@colings86).
Maintainers	Large overlap with developers. The three most active are Jason Tedor, Simon Willnauer and Jim Ferenczi (@jimczi).
Marketers	Elastic has 'Go-To-Market' partners, who help market Elasticsearch. A list of over 25 of these partners is available here.
Suppliers	Apache Lucene provides the base functionality of Elasticsearch, with Elasticsearch providing a REST API on top of Lucene, among other functionalities. Elasticsearch also runs on Apache Hadoop, Amazon Web Services (AWS), and Google Cloud Platform (GCP), thus making Apache, Amazon, and Google their main suppliers.
Support staff	Marty Messer (Vice President of Customer Care). Elastic also provides subscriptions for dedicated support for Elastic's products. Elastic's open forums handle general questions on using Elasticsearch.
System administrators	IT departments of the companies that use Elasticsearch, as well as the people who manage the Elastic Cloud platform.
Testers	Most developers are also testers. One contributor stands out in his involvement in testing as well as documentation: Luca Cavanna.
Users	A large number of users, both simple individuals as well as large companies, use Elasticsearch. Some of the largest users include Sprint, eBay, and Zalando.

Power/Interest grid

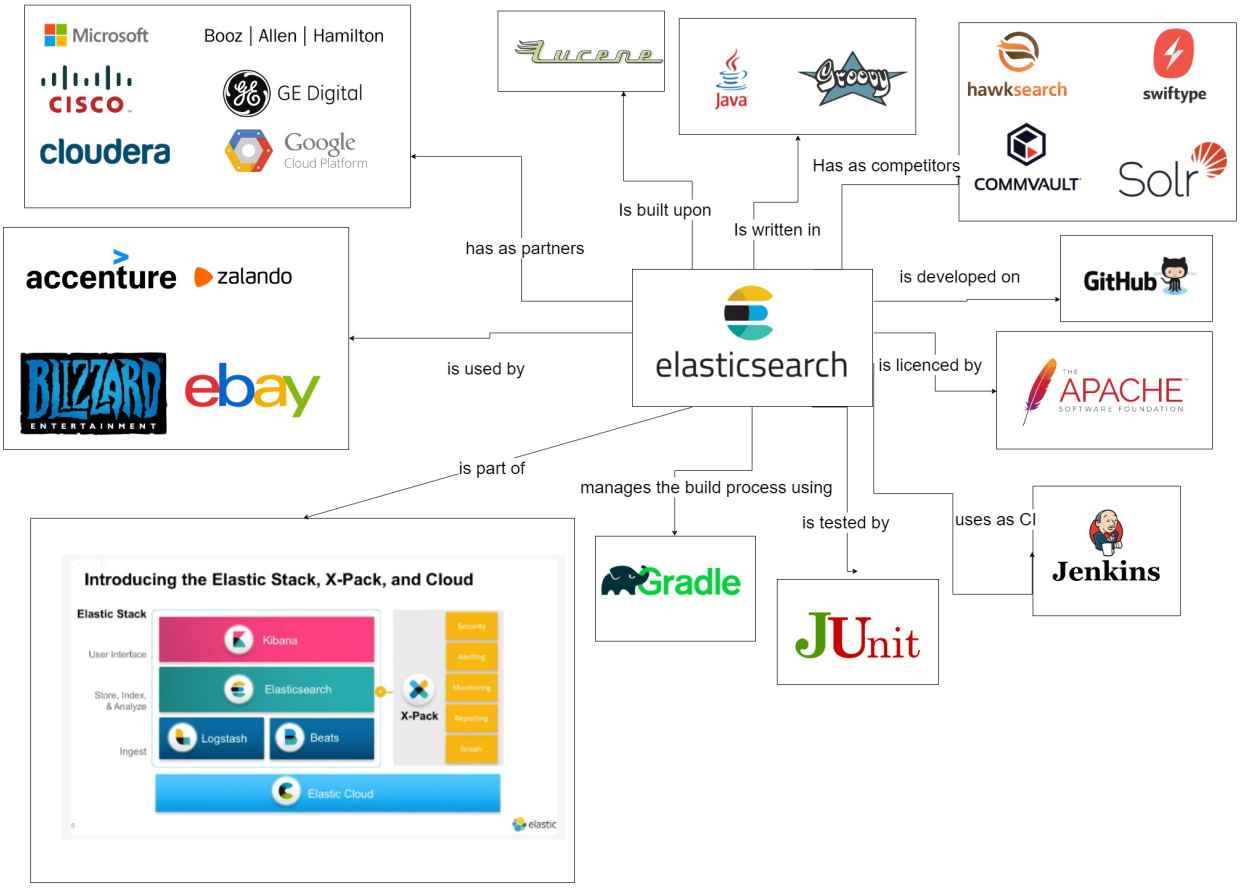
The various types of stakeholders can be placed in a Power/Interest grid, which shows the interest that each stakeholder category has in the system versus the power they have to influence the system. The grid is shown below:



Each of the stakeholder categories mentioned above has been included in the grid. We made a distinction between users *with* a subscription and users *without* one, as users *with* a subscription are able to get quick support and emergency patches if necessary, thus giving them more power over other users. While users have a high interest in Elasticsearch, suppliers do not, although they do have a high power over Elasticsearch; should they decide to stop supplying their product, then Elastic will have to adapt their products accordingly. Marketers, on the other hand, have hardly any power over Elasticsearch and are mainly interested in its core and/or most impressive features to use as selling points.

2. Context View

We have identified a number of external entities and categorized them by their involvement in Elasticsearch. This resulted in a context view, of which the general overview is discussed here. The following image shows the majority of these entities.



There are a few categories that we highlight here, the first one being the "Users" category. There are many companies that use Elasticsearch. The Elastic website alone lists 117 companies, which is not an exhaustive list. There are some large, well-known companies in this list, such as Accenture, Zalando, Blizzard Entertainment and eBay. It becomes clear that Elasticsearch can be used in many different ways. For example, eBay uses it as their search engine so customers can find the product they are looking for easily, whereas Blizzard uses it as a data analytics tool to gain insight in the large amounts of data generated by their games. Due to the extensive REST API provided by Elasticsearch, any user can connect to Elasticsearch in a way that caters to their requirements and business processes.

Furthermore, there is the "Partners" category. Elastic partners with various companies in various ways to increase their reach and market share. There are Go-To-Market partners that focus on identifying commercial customer opportunities. Furthermore, there are Technology and Platform partners which help Elastic to create more impactful and easier-to-deploy solutions based on their products. Finally, there are Original Equipment Manufacturer (OEM) partners which use Elastic products' features as part of their own product. Two major partnerships are with Google Cloud Platform, to provide Elastic Cloud on Google Cloud servers, and Cloudera, to connect Elasticsearch with Hadoop, a big data storage and processing tool.

The final category we highlight here is small, but contains one important entity. This one entity, Apache's Lucene project, is the main project on which Elasticsearch is built. It is a high-performance, full-featured text search engine library written entirely in Java. This makes it a technology suitable for nearly any application that requires full-text search, especially cross-platform applications. Elasticsearch is built upon Lucene and uses its API for data indexing and searching. Because these features lie at the core of what Elasticsearch does, Lucene is an important external entity that should be closely monitored by Elasticsearch.

3. Module Organization

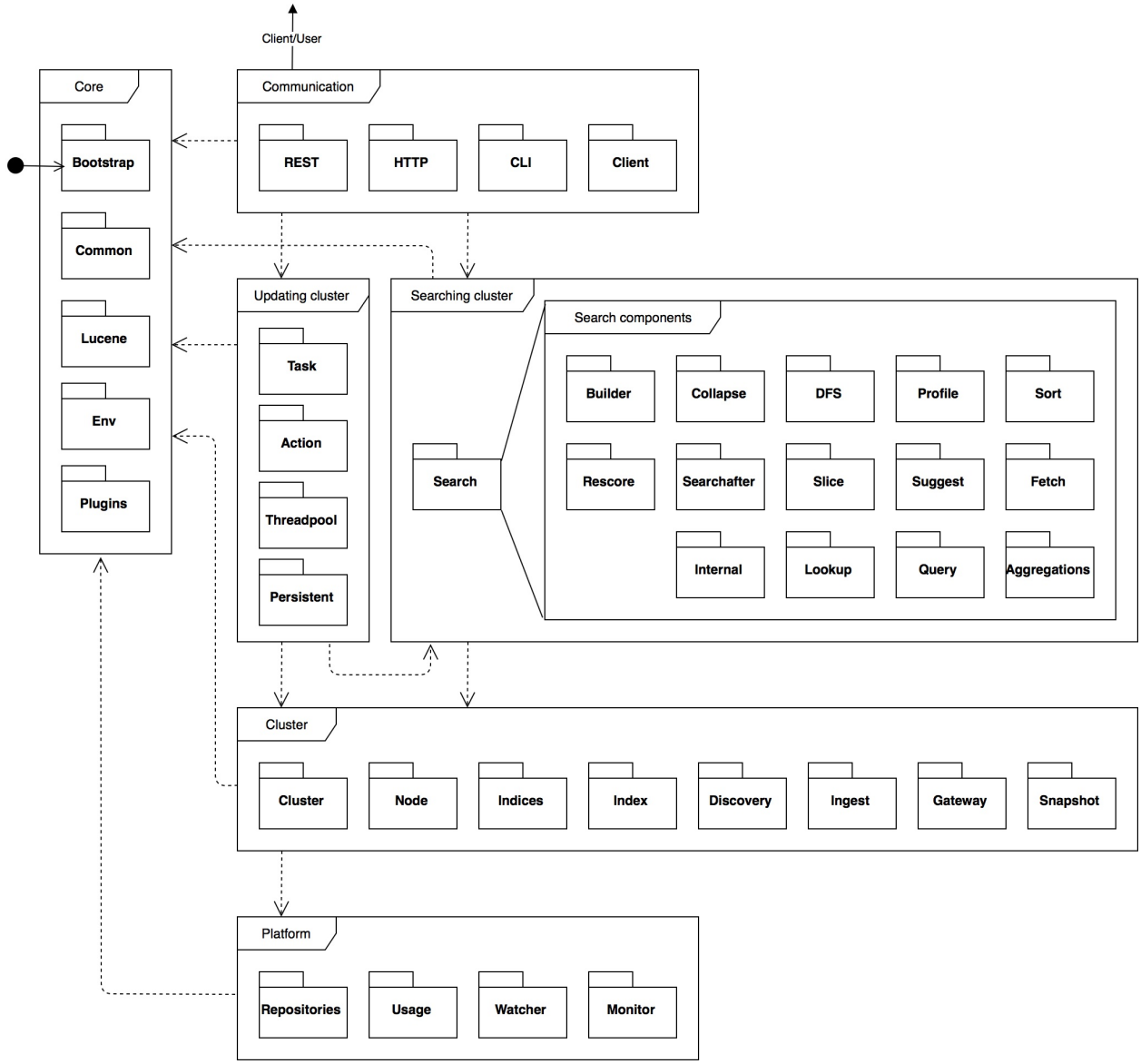
In this section, we detail the module organization of Elasticsearch, as well as important dependencies between these modules, in order to illustrate the overall internal code structure of Elasticsearch. We first discuss the `server` module of Elasticsearch, which is the module that contains the core functionality of Elasticsearch. Afterward, we take a look at the `client` module, which mainly functions as a

library to connect to an Elasticsearch cluster and send requests to it. Finally, we zoom out and take a look at how these two modules relate. The various extracted Elasticsearch `modules`, their purpose and relation to the core `server` and `client` modules are also discussed.

Do not confuse Elasticsearch `modules` with the module organization discussed in this chapter. The former represents extracted functionality from Elasticsearch that can be reused, while the latter represents a large unit in a system containing related code, as described by Rozanski and Woods (2012).

Server module

We first look at the `server` module, which contains the core functionality of Elasticsearch. The server module is structured as a large collection of packages that all provide differing functionality. We attempt to divide these packages into different *component layers*, as demonstrated by Rozanski and Woods, in order to give a clearer overview of the different parts of the system, and to avoid cluttering the component diagram with many individual packages and dependencies. The figure below shows these component layers and their relations.



We define the `core` layer, which contains packages that are either used by all other layers or serve functionality pertaining the system's runtime.

Then we define the `communication` layer, which contains all packages directly connected with the system's endpoints. This layer makes direct use of the `updating cluster` and `searching cluster` layers to answer possible user queries. These layers are discussed below.

Furthermore, we define the `cluster` layer to contain all packages related directly to building and maintaining an Elasticsearch cluster. The `cluster` layer uses the `platform` layer for storage and file system access. This layer is discussed below.

Subsequently, we define the `updating cluster` and `searching cluster` layers, which contain all packages necessary for either modifying the cluster or traversing it for results. Both these layers make heavy use of the `cluster` layer, as they operate directly on it. They also make use of the `search` package, which contains many packages related to search operations.

Finally, we define the `platform` layer to contain all packages related to low-level operations, OS related operations, or other operations requiring direct file system access.

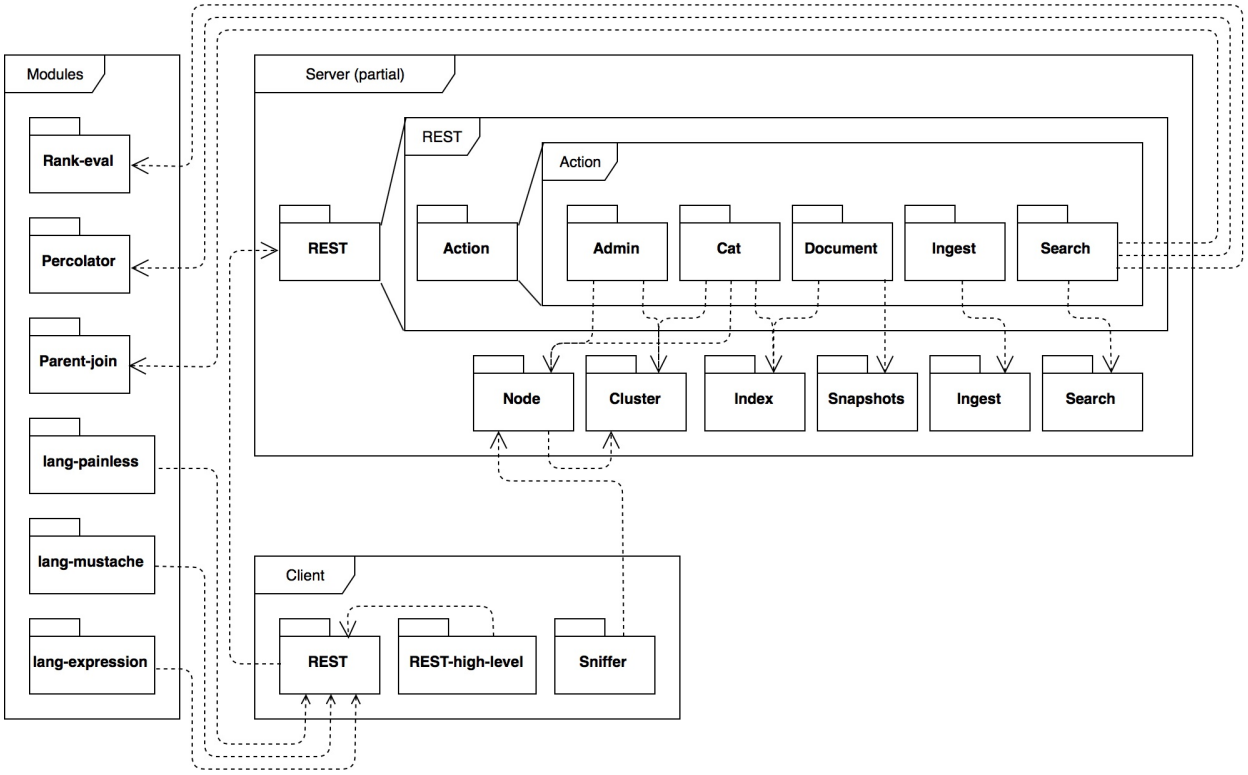
Client module

The `client` module provides an interface for other Java applications to an Elasticsearch server. From the client's perspective, the main point of entry to the server is the REST API. The `client` module contains a `rest` package which constructs REST requests and sends them to the server. To simplify the generation of REST requests, the `client` module provides a `rest-high-level` package, which wraps the lower-level `rest` package.

The `sniffer` package, meanwhile, provides clients with the functionality to automatically discover nodes by utilizing Nodes Info API, so the user does not have to manually check for new nodes in the cluster.

Client-server communication

Having discussed the structure of the `server` and `client` modules, we now zoom out and look at important package relations between them from a client's perspective. We also show how Elasticsearch has extracted functionality from `server` into Elasticsearch `modules`. The figure below shows this component diagram.



The client and server communicate through a REST API. All of these REST requests are received by the server-side `rest` package and are dispatched to the accompanying `rest.action` package. These `rest.action` packages depend on other packages in the server module, as can be seen in the component diagram. Therefore, the server and client `rest` packages represent the link between the client and the server.

In the client-server component diagram, there are several components that are not part of `server` or `client`. These are Elasticsearch modules, which are pieces of functionality that are extracted to be reusable. These modules either use one of the core packages themselves or are used either by the core server or client packages.

4. Information View

This section describes how information is handled in Elasticsearch. First, we analyze the storage model used by Elasticsearch, and how the way that data is divided increases performance and provides high availability. Secondly, we discuss data representation and how relations between entities are handled.

Storage model

The Elasticsearch storage model is based on various components, as briefly explained below.

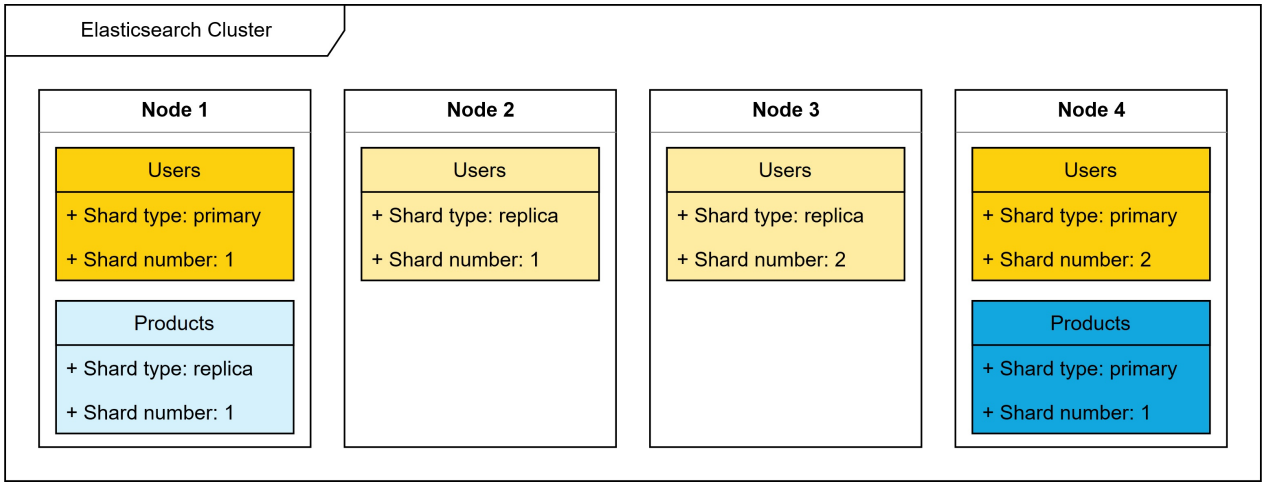
- **Document**: a basic unit of information that can be indexed. For example, this could be a document for a blog post with its creation date and content.
- **Index**: a collection of documents of the same type.
- **Node**: a group of indices that reside on a single server.
- **Cluster**: a collection of one or more nodes over which all data is distributed. It provides indexing and search capabilities across all nodes.

Elasticsearch indices are built on top of Lucene indices and are stored in a NoSQL database. More on the NoSQL nature of Elasticsearch in the next section.

An index could store such a large amount of data that it exceeds the hardware limits of a single node. To solve this problem, Elasticsearch provides the ability to divide an index into multiple pieces called *shards*. The number of shards is specified by the user. Sharding allows for horizontal splitting of volumes as well as distributing and parallelizing operations across shards, thus increasing performance and throughput.

To provide high availability, Elasticsearch also provides functionality for *replicating* shards. This means that one or more copies of shards exist on different nodes. If one node fails, all data is still available. It also results in more throughput, as searches can be executed on all replicas in parallel.

The figure below shows an Elasticsearch cluster with four nodes and two indices; a `Users` index with two shards and one replication, and a `Products` index with a single shard and a single replication. A failure of e.g. node 4 would not result in any data loss, as both shards present on that node are replicated on another node.



Data representation and relations

From a user's perspective, all data is represented in JavaScript Object Notation (JSON), a ubiquitous internet data interchange format. The data in nodes is stored using a NoSQL-like structure. Opposed to relational databases that are specifically designed to manage relationships, NoSQL databases treat the world as though it were flat. An index is a flat collection of independent documents. A single document should contain all information that is required to decide whether it matches a search request. However, relationships still matter. Therefore, there are various techniques to manage relational data in Elasticsearch, which are briefly discussed below.

- **Application-side joins**. By storing a reference to a document from a different index, e.g. an id, users can first fetch this id from one

index, and then perform another query to fetch the related document from another index. The obvious disadvantage is the need to run multiple queries.

- **Data denormalization**. Data denormalization is the process of including redundant copies of data in each document that it requires access to. For example, one can store the name of a blog post's author in both the `blog post` document itself, as well as in an `author` document. The advantage of this method is speed, the disadvantage is data duplication.
- **Nested objects**. Related entities can be stored within the same document. For example, a blog post could be stored together with all its comments, simply by passing an array of comments in the `blog post` index.
- **Parent-child relationships**. Elasticsearch provides the functionality to specify parent-child relationships between indices. This method is similar to nested objects, but the children are now separate documents.

5. Performance & Scalability Perspective

This section introduces a perspective to cover the performance and scalability of Elasticsearch. First, we recap the notion of a perspective. Then, we address the benchmark suite Elasticsearch provides and afterward, we briefly discuss concerns this perspective covers and how Elasticsearch interprets these. Thereafter, we provide an overview of the activities that Elasticsearch undertakes as part of this perspective. Finally, we discuss tactics Elasticsearch leverages in this perspective.

A perspective, as defined by Rozanski and Woods (2012), is a collection of concerns, activities and tactics used to ensure that a system exhibits a particular set of qualities, properties, or behaviors. A perspective can be applied to an architectural view of the system, to ensure that the architecture (in the context of this view) fits its purpose as defined by the perspective.

For a detailed specification of the concerns, activities and tactics that are generally part of a performance and scalability perspective, please refer to the source material.

Benchmark suite

Elasticsearch has an extensive benchmark suite, consisting of:

- The microbenchmark suite contains microbenchmarks intended to spot performance regressions in performance-critical sections.
- Elastic/Rally, a macrobenchmark framework intended to identify performance issues through (profiled) use of an Elasticsearch test cluster. This framework is periodically run against the Elasticsearch master branch, using many test clusters with different datasets. Detailed results can be found at the Elasticsearch Benchmarks website.

This benchmark suite is a useful tool for building this perspective, as is seen below.

Addressed perspective concerns

This perspective relates to a number of concerns regarding performance and scalability, which the Elasticsearch team can address with the assistance of the benchmark suite. For one, Rally records latency and service time of the system, addressing response time concerns. For another, it measures minimal, median and maximal throughput of the system at different levels of system load, addressing both throughput and peak load behavior concerns. Lastly, it measures median CPU usage during execution, allowing the team to address hardware resource requirement concerns.

While scalability of the system could be measured through the benchmark suite, Elasticsearch explicitly states that Rally does not intentionally test this. It should be noted, however, that Rally is run against large datasets.

Performed perspective activities

To address the concerns mentioned above, the Elasticsearch team appears to perform a number of activities. The team has clearly captured performance requirements, as per their website: they claim the system works for extremely large indices while providing excellent performance. Furthermore, they routinely perform practical testing, as follows from their periodically run macrobenchmark framework. Finally, Rally is run against varying sizes of data, ranging from at least two million to at least sixty million documents. As such, we see that Elasticsearch assesses their set requirements from time to time.

Leveraged perspective tactics

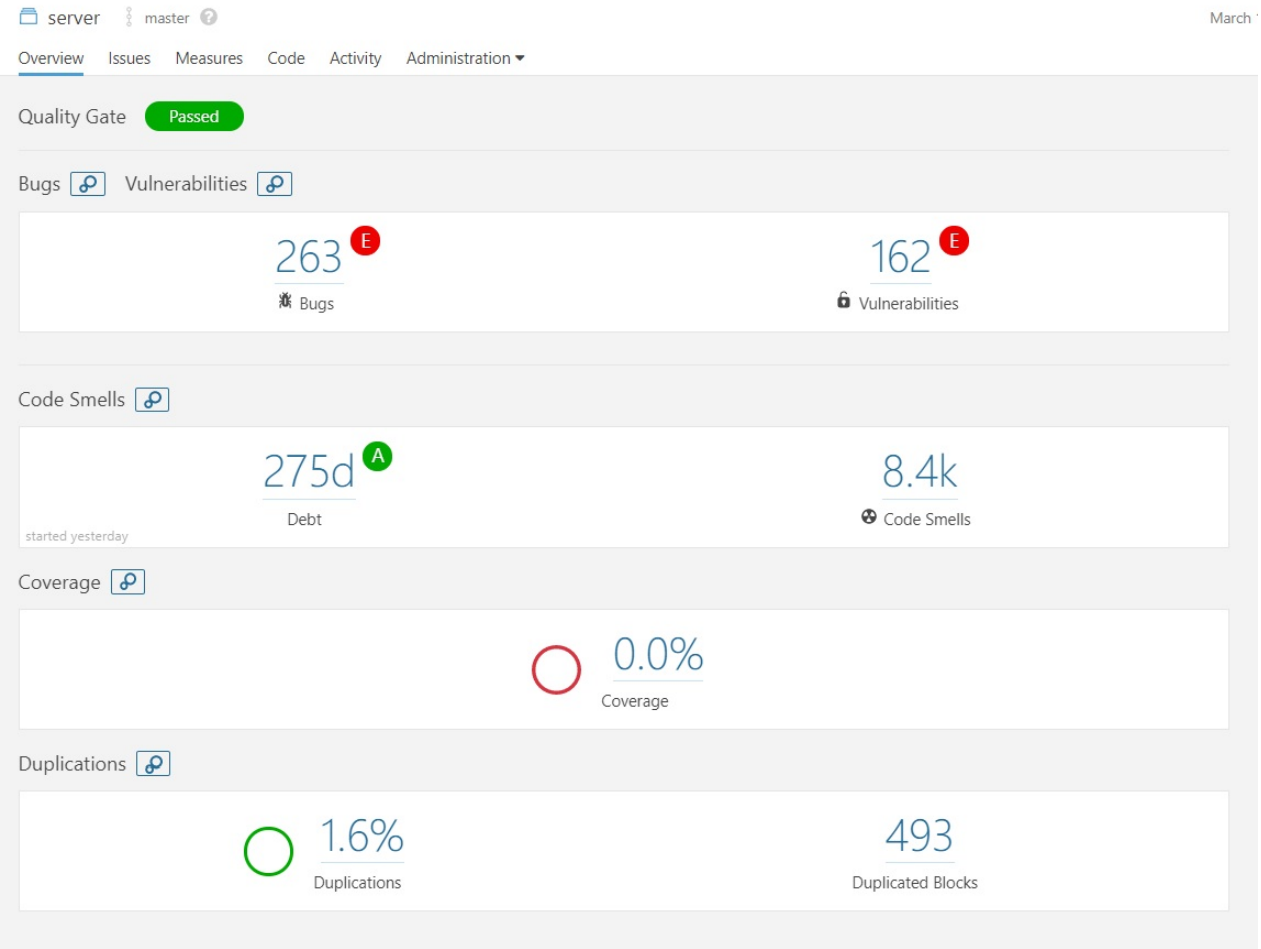
Finally, there are some tactics related to this perspective, that we see Elasticsearch leverage to improve the performance and scalability of their system. First, as Elasticsearch is by its nature distributed, it makes thorough use of asynchronous processing to reduce index operations. Secondly, Elasticsearch addresses resource contention (which happens often in distributed systems) through replication of data: multiple index nodes can share duplicates of a shard to provide fast localized access to data. Thirdly, we see that Elasticsearch focuses on optimizing often-repeated processing, evident from the microbenchmark suite, which tests relevant often-called code.

6. Technical Debt

During our research, we found that the Elasticsearch developers do evaluate the trade-off between added value and technical debt when adding new functionalities, but hardly have any discussions on the current technical debt in the system. We analyzed Elasticsearch's technical debt in several ways. First, we analyzed Elasticsearch using SonarQube. Then, we manually looked for violations of SOLID principles, went through their instrumentation and analyzed their Javadoc. Finally, we looked into their testing debt and accidentally stumbled upon some massive technical debt regarding test coverage generation.

SonarQube analysis

To analyze the project in terms of possible bugs, vulnerabilities and code smells, we opted to run SonarQube over the entire `org.elasticsearch` package of the server module, resulting in the following overview. The entire result has been made public here.



As the overview shows, SonarQube identified 280 days of technical debt, meaning that fixing all of the mentioned violations would take 280 working days, approximately 9 months. On further inspection, however, it turned out to be better than it seemed. Some of those violations were deliberate choices, while others did contribute to actual technical debt. Both are discussed in their respective subsections.

Deliberate violations

Some SonarQube violations are actually code style choices that the Elasticsearch team made during its development cycles and therefore do not constitute actual technical debt. We show two examples of this, though there are a few more of these to be found in the analysis. For conciseness' sake, these other examples will not be covered here.

The first example of this considers boolean comparison in conditionals, which is the largest contributor of violations in Elasticsearch, accounting for 1380 code smells. Boolean comparisons in conditionals between e.g. a variable `foo` and the value `false` are persistently written as `if (foo == false)`. SonarQube flags this as a code smell and suggests to write `if (!foo)` instead. Although this is a logical choice from SonarQube, as it reduces code length and therefore the chance of bugs, the choice made by the developers is also a sound one. Using the former option is more expressive than the latter, making the code more readable. This means the choice is a tradeoff between code size and readability, something SonarQube and the Elasticsearch team decided on differently.

The second example considers method and field naming conventions. SonarQube reports 132 *blocker* type issues, which are issues of the highest severity level. The Elasticsearch developers have a tendency to name `String` literals with an all-caps name, such as `String FOO`. These literals mostly represent the name of a certain field in a class, e.g. `String FOO = "foo"`. These literals are used in so-called `XContent` methods, which deal with different types (`X` types) of content that needs to be written to an output stream. Instead of hardcoding the actual strings in such an `XContent` method, the all-caps String definitions are used. This causes a number of classes to contain both the *normal* field `foo`, which is named in lower-case, as well as the `XContent` field `FOO`, named in upper-case. Although this is a deliberate choice made by the developers, SonarQube flags this as a *blocker* type issue, meaning an immediate fix is preferred.

Actual technical debt

There are also SonarQube violations that cannot be explained by the previously mentioned code style choices. Seeing as we cannot cover all of the issues flagged by SonarQube, we look at two primary categories of these violations that contain a large number of issues, as well as one easily fixable category in which we contributed towards decreasing the technical debt.

The first of these categories is collectively named *Cognitive Complexity*, which tests how many control flow statements are used within a method and how deep these statements are nested. A cognitive complexity score is calculated with these criteria and compared to a default maximum. SonarQube flags 573 occurrences of these, of which an example from `MultiGetRequest#parseDocuments` is displayed in the image below. Some of them might have a valid reason to exist, but most of them should be refactored into separate methods to reduce technical debt.

```
while ((token = parser.nextToken()) != Token.END_OBJECT) {
    if (token == Token.FIELD_NAME) {
        currentFieldName = parser.currentName();
    } else if (token.isValue()) {
        if (INDEX.match(currentFieldName, parser.getDeprecationHandler())) {
            if (!allowExplicitIndex) {
                throw new IllegalArgumentException("explicit index in multi get is not allowed");
            }
            index = parser.text();
        } else if (TYPE.match(currentFieldName, parser.getDeprecationHandler())) {
            type = parser.text();
        } else if (ID.match(currentFieldName, parser.getDeprecationHandler())) {
            id = parser.text();
        } else if (ROUTING.match(currentFieldName, parser.getDeprecationHandler())) {
            routing = parser.text();
        } else if (PARENT.match(currentFieldName, parser.getDeprecationHandler())) {
            parent = parser.text();
        } else if (FIELDS.match(currentFieldName, parser.getDeprecationHandler())) {
            throw new ParsingException(parser.getTokenLocation(),
                "Unsupported field [fields] used, expected [stored_fields] instead");
        } else if (STORED_FIELDS.match(currentFieldName, parser.getDeprecationHandler())) {
            storedFields = new ArrayList<>();
            storedFields.add(parser.text());
        } else if (VERSION.match(currentFieldName, parser.getDeprecationHandler())) {
            version = parser.longValue();
        } else if (VERSION_TYPE.match(currentFieldName, parser.getDeprecationHandler())) {
            versionType = VersionType.fromString(parser.text());
        } else if (SOURCE.match(currentFieldName, parser.getDeprecationHandler())) {
            // check lenient to avoid interpreting the value as string but parse strict in order to provoke an error early on.
            if (parser.isBooleanValueLenient()) {
                fetchSourceContext = new FetchSourceContext(parser.booleanValue(), fetchSourceContext.includes(),
                    fetchSourceContext.excludes());
            } else if (token == Token.VALUE_STRING) {
                fetchSourceContext = new FetchSourceContext(fetchSourceContext.fetchSource(),
                    new String[]{parser.text()}, fetchSourceContext.excludes());
            } else {
                throw new ElasticsearchParseException("illegal type for _source: [{}]", token);
            }
        } else {
            throw new ElasticsearchParseException("failed to parse multi get request. unknown field [{}]", currentFieldName);
        }
    } else if (token == Token.START_ARRAY) {
```

The second category deals with superfluous code. SonarQube finds 117 occurrences of unused method parameters and 237 occurrences of unused import statements. Keeping code clean is important, as this leads to more readable code, which leads to an easier development process. Cleaning up code like this is one of the simpler tasks that can be found in this project, but this still requires some more work to be done.

The final category deals with issues in `equals(Object other)` methods: these should test the argument's type before casting it. Also, once cast, some of these `equals` methods actually compare identical expressions. For example, if an object has a field `foo`, then the method would compare the equality this field using `Object.equals(foo, foo)` instead of `Object.equals(foo, other.foo)`. Both of these issues are easily fixable, but can cause serious bugs, so we decided to fix all instances of both these violations in a pull request.

SOLID violations

We also identify two examples of SOLID violations in the Elasticsearch codebase, for which we propose a solution if deemed possible. For conciseness' sake, we do not cover the Liskov Substitution, Interface Segregation and Dependency Inversion principles here.

Single Responsibility Principle

First off, the Single Responsibility Principle is violated in `org.elasticsearch.index.shard.IndexShard`. This class represents a *shard*, as explained in the information view. To our knowledge, the class is responsible for at least the following:

- Creating, storing, updating and searching through a single index inside the shard.
- Recovery of the shard and its index after a restart.
- Read/write operations between heap and file system concerned with indices.
- Error recovery when some operation on the index fails.

This class is imported into other classes 97 times, spans over 2600 lines of code, and its constructor uses 18 parameters. Given that it carries many responsibilities, is coupled to a major part of the codebase and is so large, we also consider this class a prime example of the *God-Object* anti-pattern.

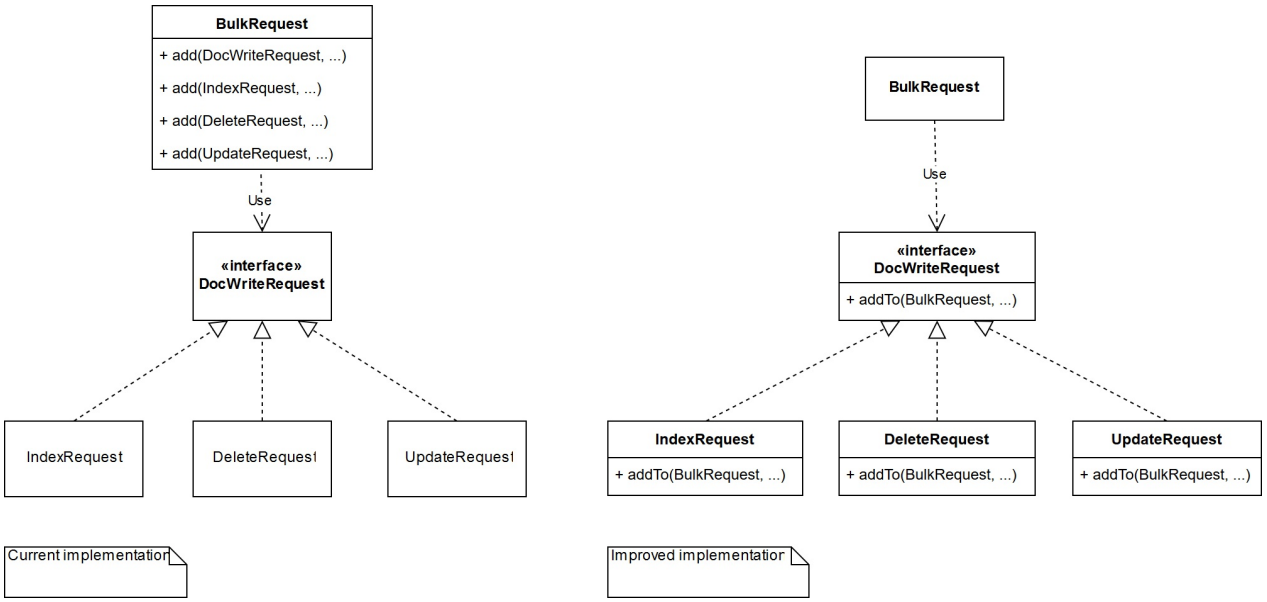
As such, modifying this class to adhere to the Single Responsibility Principle becomes complex, requiring modifications to almost the entire codebase, making such a modification almost unattainable.

Open-Closed Principle

Secondly, we identify a violation of the Open-Closed Principle concerning `org.elasticsearch.action.BulkRequest` and `org.elasticsearch.action.DocWriteRequest`. Specifically, the `BulkRequest#add(DocWriteRequest, Object)` method (shown in the figure below) must be expanded if ever a new type of request is to be supported by `BulkRequest`, which implies that `BulkRequest` is not closed for modification.

```
public BulkRequest add(DocWriteRequest request, @Nullable Object payload) {
    if (request instanceof IndexRequest) {
        add((IndexRequest) request, payload);
    } else if (request instanceof DeleteRequest) {
        add((DeleteRequest) request, payload);
    } else if (request instanceof UpdateRequest) {
        add((UpdateRequest) request, payload);
    } else {
        throw new IllegalArgumentException("No support for request [" + request + "]");
    }
    indices.add(request.index());
    return this;
}
```

A solution would be to move this functionality into the respective `DocWriteRequest` child classes by adding an `addTo(BulkRequest, ...)` method to the `DocWriteRequest` interface, as shown in the figure below.



This way, the `BulkRequest` class does not have to be extended whenever `BulkRequest` needs to support a type of request. The functionality is simply pushed to the new child class (open for extension), while the `BulkRequest` class remains the same (closed for modification).

Instrumentation

Lack of instrumentation is also a form of technical debt. We found that Elasticsearch does miss some important static code analysis tools, such as PMD and FindBugs, and does not make optimal use of CheckStyle.

Elasticsearch uses Gradle as their build system and allows users to execute the static code checks using `./gradlew precommit`. They employ CheckStyle to verify code style conventions, although their CheckStyle configuration is not very extensive. They only check a small number of rules, such as avoiding star imports, naming files in compliance with the classes they contain, and avoiding empty Javadoc comments. However, they do not check if there are any Javadoc comments at all, nor do they check method length, class length, method and class complexity, or proper indentation. What is even more interesting is their check for line length: their CheckStyle configuration defines a maximum line length of 140 characters, but the colocated CheckStyle suppressions file suppresses this check for every file that does not pass it until these files start to pass the check, as stated in the documentation.

Javadoc

Looking through the codebase, we noticed that many methods and classes are missing Javadoc. We happened to find this issue in which a community member indicates the same. An Elasticsearch member responds by saying that he is happy with the current situation regarding Javadoc, also seeing as they had "almost no Javadoc 4 years ago". He makes it clear that they have recently started being "much more diligent about adding Javadocs to public methods whenever we add them or touch existing ones, also enforcing that habit as part of the review process". While this does show that there are some efforts being made to reduce Javadoc debt, it also becomes clear that the main focus is to keep the number of uncommented methods as small as possible in the *public* API, indicating that the Elasticsearch developers focus on the *user* experience first and on the *developer* experience second.

Testing debt

Especially for large and distributed projects such as Elasticsearch, it is important that software is properly tested. Weak tests or low test coverage are therefore also part of the technical debt of a system. This section identifies a part of the testing debt present in Elasticsearch. First, we look at the testing procedures; how to run the tests, as well as how to write them. Then, we analyze the test coverage, or rather the lack thereof.

Testing procedures

The Elasticsearch repository contains a TESTING.asciidoc file that explains how to use Gradle to compile Elasticsearch and run its tests.

Unless one is specified, a randomized testing seed is generated each time the tests are run. This enables randomized testing, a black-box testing technique where test inputs are randomly generated, so over time a large number of possible test inputs, if not all, are fed to the class or method under test. This may cause flaky tests, but mainly reduces the chance of bugs being overlooked due to a developer's misplaced trust in the code and is easy to set up.

Elasticsearch also contains a testing framework used throughout their tests, holding a number of abstract test case classes as well as a large number of utility classes that aid in creating mocks of certain parts of Elasticsearch. Such a testing framework provides a good basis for creating concrete test cases, creates consistency between testing environments for different tests and minimizes code duplication across test suites.

The basis for all Elasticsearch test cases is the `ESTestCase` class, which contains a number of methods for initializing and cleaning up test cases, some convenience methods for generating random inputs, custom assertions, and a host of other utility methods. The testing framework also contains other such `TestCase` classes extending from `ESTestCase`, each providing additional setup and cleanup, or more specific convenience methods and custom assertions to be used in specific tests. Elasticsearch even has tests for the more complex parts of their testing framework.

Test coverage generation

The testing documentation stated until recently that a test coverage report could be generated using `mvn -Dtests.coverage test jacoco:report`. However, this is a Maven command, while Elasticsearch migrated to Gradle in October 2015. As this documentation was not updated since, this may indicate that the developers do not pay much (if any) attention to their test coverage.

We notified them of this, to which they replied that JaCoCo is not included in their build configuration. Applying JaCoCo is usually as simple as adding `project.pluginManager.apply('jacoco')` to the building code, but this did not work. Upon further investigation, we found that the Elasticsearch build system replaces each Gradle `Test` task with their own `RandomizedTestingTask` in order to enable randomized testing. While this class should extend Gradle's `Test` task to be recognized as a testing-related task, it instead extends the `DefaultTask` class. Seeing as JaCoCo binds to instances of `Test`, JaCoCo is unable to find any tasks to bind to. An Elasticsearch developer did inform us that he originally wanted `RandomizedTestingTask` to extend `Test`, but that there were many issues with it. After more experimenting, we concluded that `RandomizedTestingTask` would need a significant rewrite by someone with sufficient experience so as to keep the required existing functionalities, though we did update the outdated documentation.

Having a build configuration unable to support testing instrumentation such as JaCoCo is a major technical debt, as it also causes problems upon integrating other instrumentation, such as SonarQube or mutation testing tools.

7. Conclusion

This chapter provided an overview of the software architecture of Elasticsearch. We identified different stakeholders, and used several architectural viewpoints to gain insight into the system. We also identified and analyzed the technical debt in the Elasticsearch codebase, both from a code quality and testing perspective.

We can draw various interesting conclusions. First, we believe that the different modules in the Elasticsearch codebase are well organized. Each package has a clear responsibility, and the extraction of reusable modules provides a good separation between core and additional functionality.

The Elasticsearch team also has a good focus on performance and scalability. There is an extensive benchmark suite, which is also used in GitHub discussions when making merge decisions.

However, there are also some issues present in the system. There is a substantial amount of technical debt present in the system. Furthermore, there are some fundamental issues with the build system, which prevents generating test coverage reports. The main focus of the core developers lies on user experience, rather than developer experience.

References

1. Rozanski, N., & Woods, E. (2012). Software Systems Architecture (2nd ed.). Pearson Education.

Electron | Build cross-platform desktop apps with JavaScript, HTML, and CSS

By Shivam Miglani and Sharad Shriram, Delft University of Technology.

Abstract

Electron is an open-source software framework that was initially developed by Cheng Zhao, an engineer at GitHub for building native desktop GUI applications with web technologies like JavaScript, HTML, and CSS[1]. This project is developed using node.js runtime as back-end and chromium as the front-end. Several popular and widely-used open source projects like Atom, Slack, Skype, VS Code, Github Desktop, and about 500 more native desktop applications are built on Electron. This chapter summarizes the architecture of Electron through stakeholders involved and their interests, different views ranging from development to deployment, and perspectives such as evolution and security.

Table of contents

1. What is Electron?

Developing desktop GUI applications is laborious and it requires Operating System (OS) specific expertise, which leads to additional cost and time overhead along with an increase in the technical complexity of the project. For such projects, maintaining the quality, functionality, and security across multiple operating systems is a difficult task. Electron provides a solution to this problem by providing a framework for developing cross-platform native desktop applications using web technologies like JavaScript, HTML and CSS. This means that a skilled web developer can develop such cross-platform applications without the need of breadth of OS-specific skills.

The applications built from Electron consist of two types of processes: the main process and multiple renderer processes which follow a *master-slave* pattern. The main process of the application is the master and the multiple renderer processes are the slaves which communicate using Inter-Process Communication (IPC). Fig. 1 shows that this is analogous to a Chromium browser where there is one main window and multiple tabs (renderer processes).

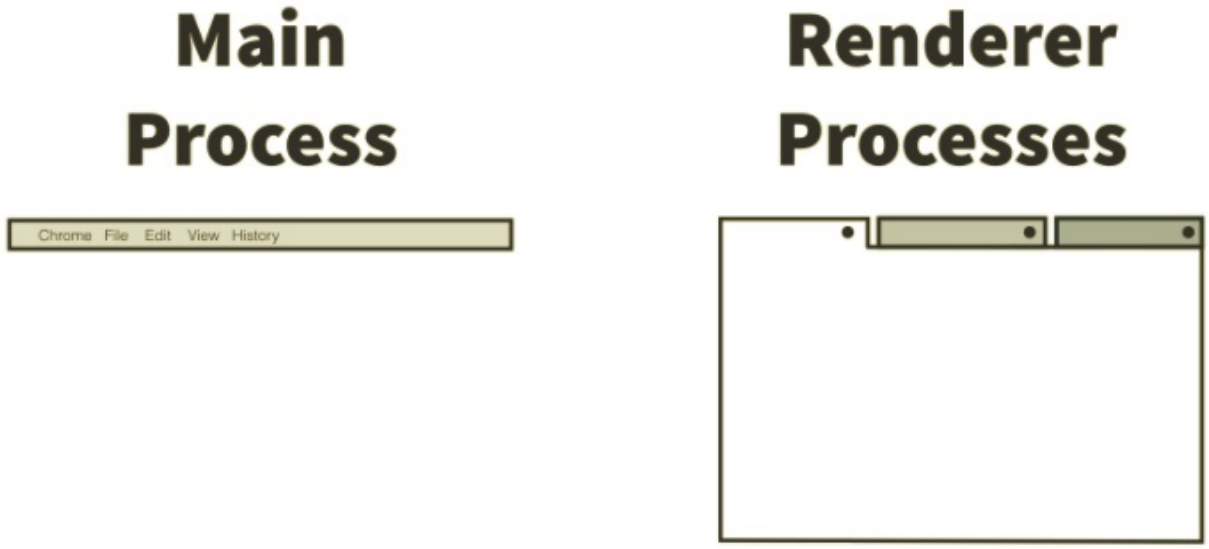


Figure 1: One Main process with Multiple renderer processes. This is analogous to a window of Chromium browser with multiple tabs

Specifically, Electron looks into the start entry in the `package.json` manifest included in the project to determine the entry point of the application, which runs as the main process.

- *Main process*: The main process is responsible for responding to application lifecycle events such as starting up, quitting, preparing to quit, going to the background, coming to the foreground, and more. The main process is also responsible for communicating to native operating system APIs. For example, to display a dialog to open or save a file. [2] The main process can also create and destroy renderer processes using Electron's `BrowserWindow` module.
- *Renderer processes*: Renderer processes can load web pages to display a GUI. Each process takes advantage of Chromium's multi-process architecture and runs on its own thread. Unlike normal web pages, there is access to all the Node APIs in the renderer processes, allowing developers to leverage native modules and lower-level system interactions. [2]

At the time of writing, the stable version of Electron is 1.8.4 and all our analysis are mainly focused on the `master` branch of core Electron's repository. Before doing in-depth architectural analysis let's look at the people and organizations involved in Electron's development.

2. Stakeholder Analysis

Stakeholder Analysis describes the people involved in Electron and their roles. We have categorized them according to the eleven types proposed by Rozanski and Woods[3].

Type	Stakeholders	Description
Developers	Core developers, Committer team and contributors	Core developers like @zcbenz, @zeke and others set policies and are the general managers. Most of the core developers work at Github (which sponsors Electron). Committers like @deepak1556 and open-source active developers like Alexey Kuzmin from Microsoft, Felix Rieseberg from Slack help the core developers with pull requests.
Acquirers	Core Developers at Github	The core developers and (some) senior contributors from prominent users (Alexey Kuzmin from Microsoft and Felix Rieseberg from Slack) decide the future roadmap for Electron.
Assessors	Developers	Developers of Electron as well as from the user organizations using Electron assess the conformance to standards and legal regulations themselves.
Communicators	Teachers	Electron is taught by teachers in online environments such as YouTube Videos, Tutorials and Blogs on the web.
Maintainers	Core developers, Contributors	The overall evolution is maintained by the core developers, and maintenance tasks by both core developers and contributors.
Production Engineers	John Kleinschmidt and team	Manages the production releases and runs tests on new builds. He also manages the code for Circle-CI integration tests.
Suppliers	Github	Github sponsors Electron and is a supplier for Electron as it provides software and infrastructure for Electron to run.
Support Staff	Developers, Teachers	Support for development and contributions for Electron is done through mailing lists, Slack, Discuss Forums and platforms such as StackOverflow.
System Administrators	John Kleinschmidt and Core developers	They control the evolution and development of the project. They also manages the code for integration tests through Circle-CI, Travis-CI, AppVeyor etc.
Testers	Developers, Committer team	They are responsible for the testing of new commits and builds through explicit unit and application tests.
Users	Developers and organisations using Electron	Electron is used in Skype, Slack, WhatsApp, Atom close to 7400 independent developer forks.

Going beyond Rozanski and Woods classification:

The following stakeholders identified are additional stakeholders who do not match the groups in Rozanski and Woods[3].

- *Other repositories and products in Electron Umbrella project* like Spectron the testing framework, i18n the home for documentation and its translations, electronjs.org the website, libchromiumcontent the content-rendering library of Chromium, node the backend component, devtron the chrome development tools and around 40 other repositories complete Electron as a framework.
- *End-Users* are normal users who experience a native desktop app built on Electron and it indirectly influences Electron's development.
- *Bloggers* are unambiguous and reliable sources of information who use and help new developers to use Electron.
- *Translators* contribute to translations for Electron's documentation to different languages for its website and there are close to 300 translators (including us) who have contributed for this purpose till date.

Quantifying the stakeholder's involvement: Power vs Interest Grid

Mendelow's power/interest grid[5] is used to classify the groups of stakeholders necessary to be managed closely. Figure 2 shows that the core developers, committer team, active contributors from prominent products like Skype, Slack, etc. using Electron are stakeholders who have both high interest and power. These are the people that actively contribute to and maintain the project and need to be managed closely. Teachers and projects that use Electron without active contributions show high interest but have low power and must be informed well. Electron's dependencies have low interest and low power. The power of Chromium is slightly higher because it influences Electron's future developement as explained in the development viewpoint.

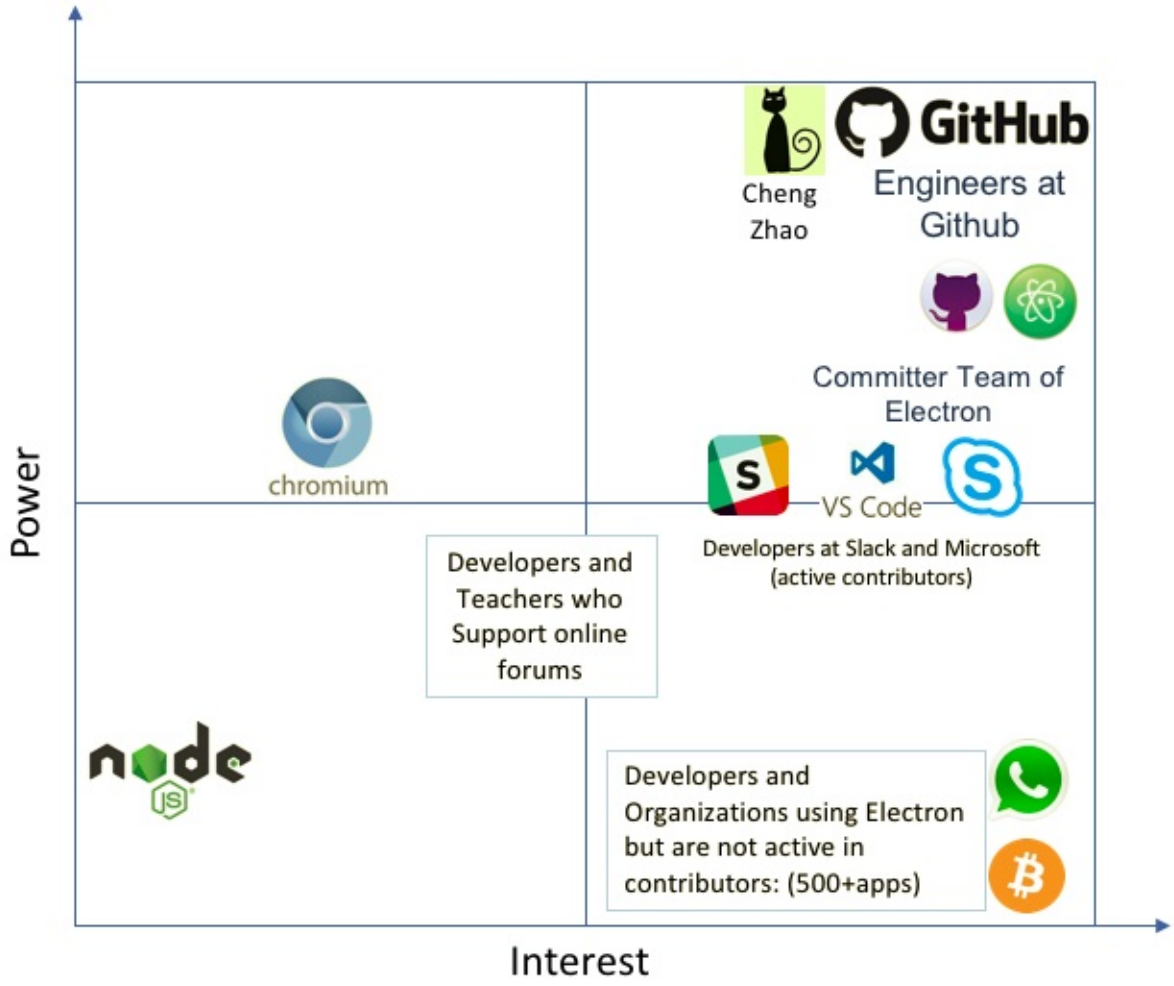


Figure 2: Power Interest Grid of Electron where the interest of the stakeholder is shown on the horizontal axis and the power of the stakeholder is shown on the vertical axis.

3. Architecture

In this section, we describe the architecture of Electron based upon various (1) *views* and (2) *perspectives*. According to Rozanski and Woods [3], a view is a representation of one or more structural aspects of an architecture that illustrates how the architecture addresses one or more concerns held by some stakeholder and an architectural perspective is a collection of activities, tactics, and guidelines that are used to ensure that a system exhibits a particular set of related quality properties that require consideration across a number of the system's architectural views.

3.1 Views

In this section, we describe the context, development and deployment viewpoints of Electron.

3.1.1 Context Viewpoint

The context view describes the scope and responsibilities, relationships, dependencies and interactions around Electron [3], as shown in Figure 3.

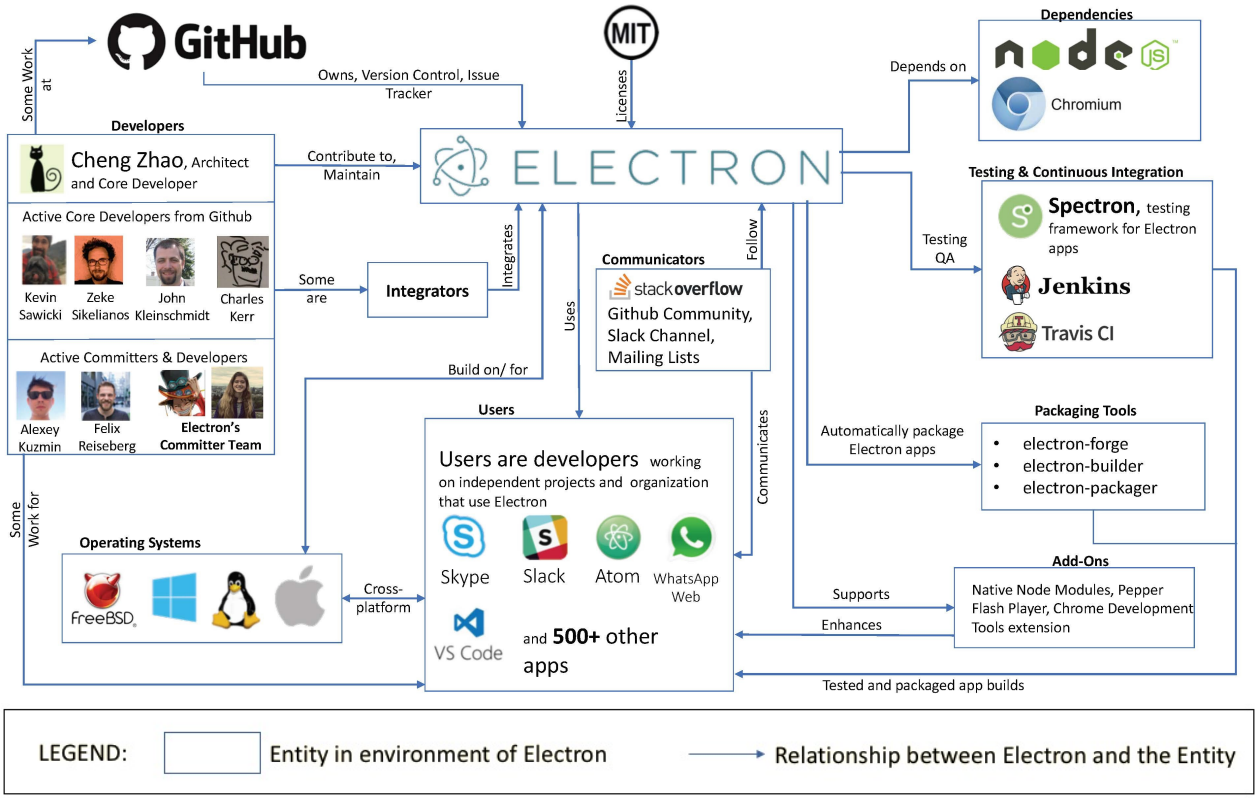


Figure 3: Context View of Electron showcasing relationships with its environment

Some prominent findings from the context view diagram are:

- **Integrators** are the active core developers from Github. The are the architects of the Electron and their merging their challenges and merging strategies are explained in the Evolution Perspective
- **Users** are organizations and developers of cross-platform desktop applications using Electron. The **communicators** are the link between users and Electron's developer community.
- **Testing Frameworks and Continuous Integration**: Spectron is the testing framework for Electron apps. Travis and Jenkins are the CI systems for testing new application build using the virtual display driver for Chromium.
- **GitHub** sponsors the entire project and provides infrastructure for version control using git and issue tracking with Github issue tracker.
- **Target Platforms**: The Electron apps are built on/for MacOS, Windows and Linux platforms. The **Packaging tools** allow for automatic packaging of Electron apps for these platforms.
- **Add-ons** like Native Node modules, Pepper Flash plugin etc. are supported by Electron and enhance the features of developed applications.

3.1.2 Development Viewpoint

Electron combines Chromium's *single thread multi-process* model[18] and Node's *single thread event-loop*[13] model into **single runtime**. As a consequence, Electron design patterns resembles its two primary dependencies namely Node.js and Chromium Webkit. Electron includes customized forked embeddings of Chromium and Node modules to keep its size small. Electron application's design pattern is similar to the *master-slave* pattern as discussed in section 1.

The *layered architecture* shown in Figure 4 gives a very high-level overview of Electron and applications built using Electron. In this section, the development view of Electron's core is summarized using module organization and dependencies, common design models and codeline models

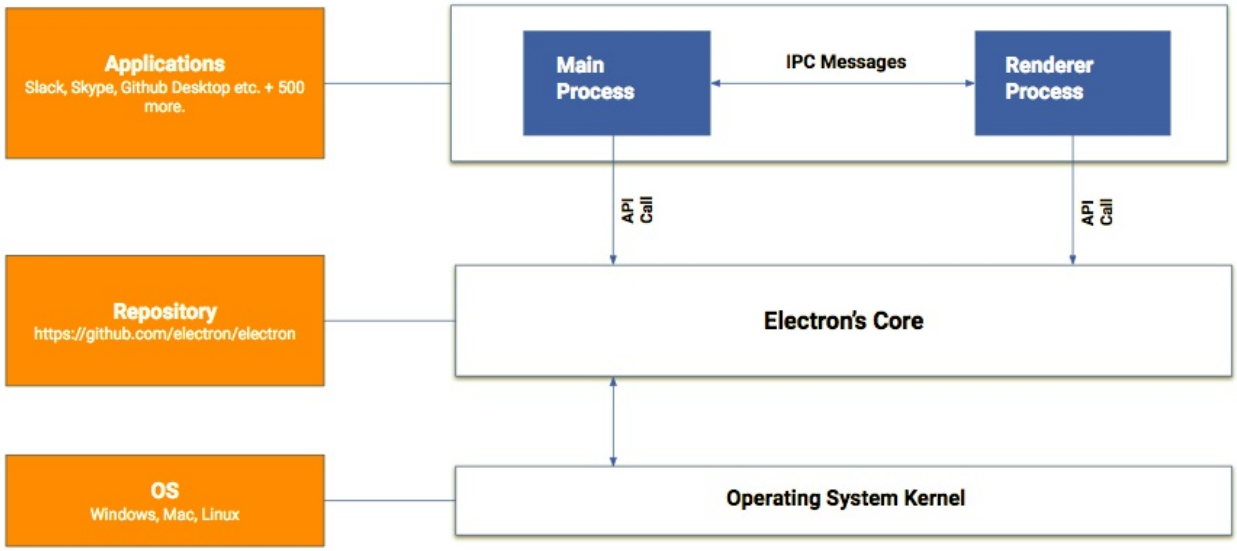


Figure 4: Layered Architecture of Electron

Module Organization

Apart from reducing size, the customized forked embeddings of Chromium and Node removes the need for external interfacing dependencies, and improves speed and performance through preconfiguration of dependencies. The module organization in Figure 5 demonstrates **high cohesion** through *Electron Core block* and **low coupling** as demonstrated through custom embeddings of the dependencies in *Electron extensions*, which is an example of good software design.

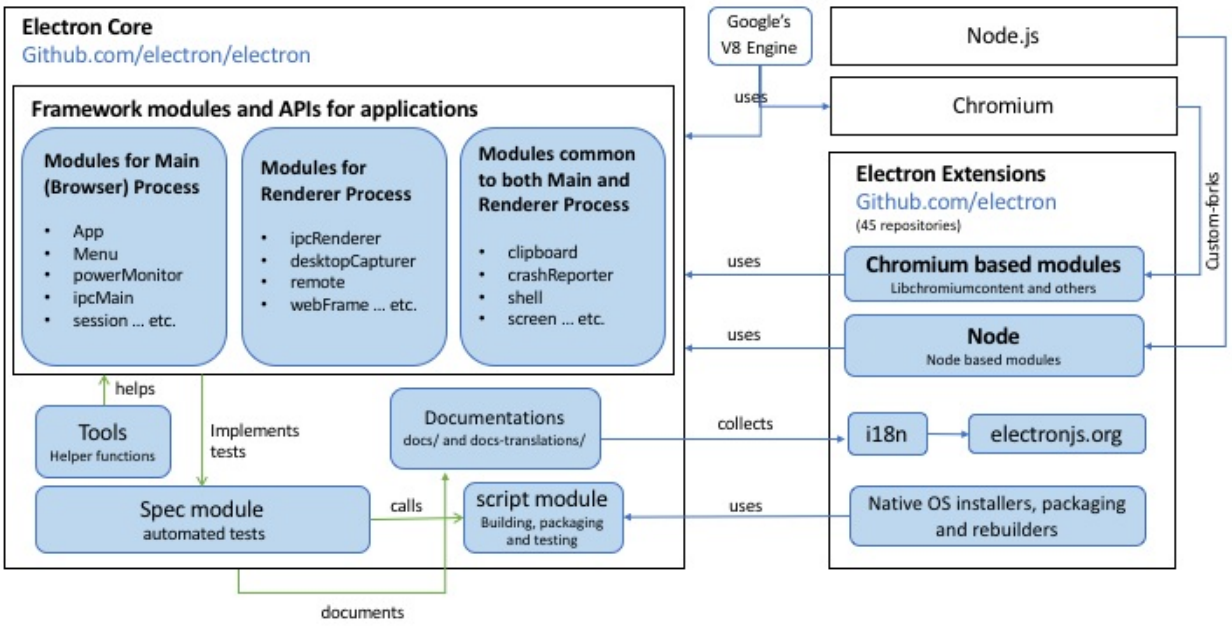


Figure 5: High-level module organization of `electron/electron` *repository and the other 45 repositories where the links to external dependencies are shown in color* blue *whereas internal dependencies in Electron's core are shown in* green.

Now we explain the 3 blocks in the diagram:

- **V8 engine**: Electron, Node and Chromium use V8, an open-source engine which converts JavaScript directly into native machine code with major optimizations. The V8 engine optimizes the code at runtime based on code heuristics and employing different optimization strategies like inline caching.

- **Electron extensions:**
  - Electron embeds different Chromium modules like `libchromiumcontent`, the content-rendering module. Other examples are `native-mate` which makes writing JS bindings easy, and automatically converts V8 types to C++ types, and `pdf-viewer` is a fork of Chrome's pdf extension to work as `webui` page, etc.
  - The custom embedding of Node runtime environment handles the interactions between with the native OS and various add-on node packages. Electron also uses some node packages for internal use like `node-minidump` is used to process minidump files, and the full list can be seen in [10].
- All of the Electron's documentation and translations are collected in i18n repository as a large JSON object in every language. The JSON object from i18n repository is used as content in the electronjs.org website.
- *Native OS installers* are modules developed for different native operating systems on which Electron can be installed like `windows-installer` for the Windows operating system. Electron apps can be *packaged* to `Asar` format which is a tar like archiving format providing advantages of mitigating issues like long file names in Windows, speeding up `require` and concealing source code from cursory inspection.[21]. Electron supports native Node.js module injection using `electron-rebuild` which *rebuilds* native them against the currently installed Electron's version.
  - **Electron's Core:**
  - *Framework Modules and APIs for applications*: In applications, Electron APIs can be used for both the main process and renderer process. Node APIs are available globally, while only DOM/Browser APIs are available in a renderer. Based on Figure 6 and the API documentation [11], the following is an overview of some common APIs used:
    - **Main process:**
      1. `app` control application's event lifecycle.
      2. `ipcMain` for asynchronous communication from the main process to renderer processes.
      3. `autoUpdater` enables automatic updates for apps
      4. `session` maanage browser sessions, cookies, cache, proxy settings, etc.
    - **Renderer process**
      1. `ipcRenderer` for asynchronous communication from a renderer process to the main process.
      2. `remote` remote invocation of methods in the main process.
      3. `webFrame` customises the rendering for the current web page
    - **Common APIs**
      1. `crashReporter` submits crash reports to a remote server.
      2. `clipboard` performs copy and paste operations on the system clipboard.

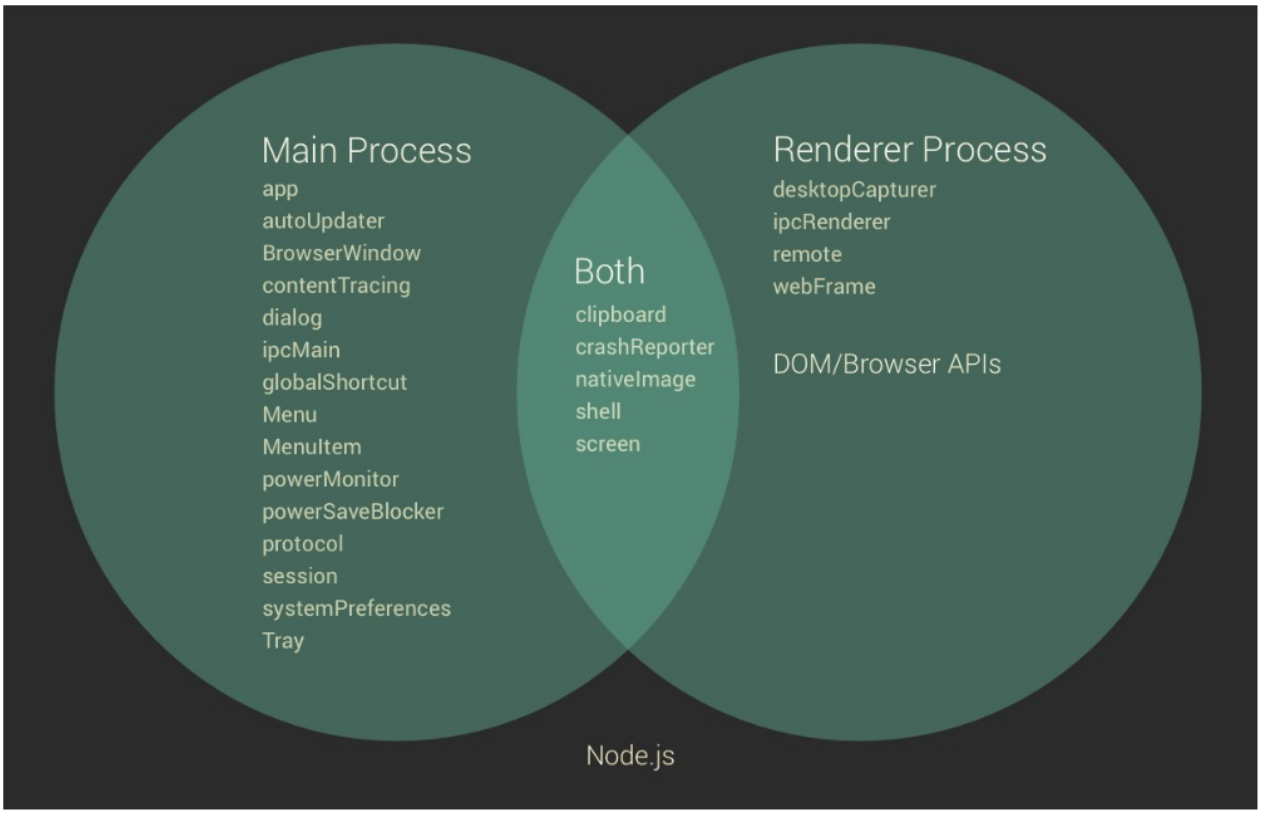


Figure 6: Venn Diagram of Electron module APIs to be used by developed application's main and renderer process. The universal set is other node.js module APIs which are available globally.

- *Script,spec and tools modules*: `script` contains scripts used for development purposes like building, packaging and testing. `spec` contains automatic tests for the framework APIs (main, renderer and common), packaging formats (asar), version checks etc. `tools` contains non user centric helper functions. The framework (API) modules use `tools` as helper functions and `spec` for creating automatic tests.

Common Design Models

Commonality across the different versions of Electron is done by defining a set of strict design constraints that are followed in Electron's development.

- *Common Processing*: Based on [3], the common processing models for Electron combines the processing models used in the framework architecture and the application architecture
  - *Termination and restart of operation* on the framework during application development is with the `main process`. For applications, the app API that controls the application's event lifecycle is responsible for the termination and restart operations.
  - *Message Logging* - `crashpad` is a Chromium project used for capturing, storing and transmitting postmortem crash reports from a client to an upstream collection server for diagnostic purposes[14, 15]. For applications, communication between the main and renderer processes are done as IPC messages using the `ipcMain` and `ipcRenderer` API modules.
  - *Internationalization* - Electron's developer community is worldwide and thus Electron's documentation is translated into different languages under the Electron-i18n project. Moreover, to facilitate better contribution and maintain a supportive, active community the conduct of conduct states,

    > In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

- *Internal and external interfacing* in Electron is applicable to both the framework and application APIs. In the framework, the different code modules and method invocations between the framework are internal interfacings whereas the Node and Chromium modules used the account as external interfacing. In applications, the API bindings between the main process and renderer processes are internal whereas additional Node packages that are used on applications as add-ons are the external interfacing.

- *Standardization of Design*: Electron's development guide documents the steps for issue creation, contributing to issues, making pull requests in accordance to the conduct of conduct which is adhered by every developer and contributor. This section briefly summarizes the design standards laid out for Electron.

  - *Issues* is the first stop for either a developer, contributor or a user to get technical assistance with build-specific issues like compilation errors, display errors, system crashes, etc. The issues found on the repository of Electron, are generally resolved within 3-7 days [6].

  - *Pull Requests* facilitate the contribution to addition of new features, modification of existing features or bug fixes for a particular feature of Electron. There is an official documentation for pull requests which standardizes the steps to be followed for submitting a new pull request to Electron. Every pull request is subjected to a Continuous Integration (CI) check and the merge pattern involves thorough code review and discussion with contributors.

  - *Coding Styles* standardizes formatting the source code making it easy to read and understand by the contributor community.

    1. Electron is mostly written in C++ with some flavours of Objective-C codes which is styled based on Chromium's C++ Style Guide. Electron uses the `clang-format` for formatting C++ source code.
    2. Electron uses Python version 2.7 as per the documentation for generating projects, follows the Chromium Coding Style and uses `script/cpplint.py` script to check code formatting to conformance.
    3. Electron is well documented and is written entirely on Markdown with the Github style. A node package called `electron-docs-linter` developed to ensure that documentation changes are formatted correctly and can be used as `npm run lint-docs`
    4. JavaScript engines used in Electron are written in the standard style with the newer ES6/ES2015 syntax where appropriate.
    5. Naming Conventions for files and variables in the source code is a good design practice which makes the code base consistent and stable for continuous development. In Electron, file names should be concatenated with `-` instead of `_` and naming variables in code is similar to Node.js as Electron APIs uses the same capitalization scheme.

- *Standardization of Testing*: Testing in Electron can be done in the framework and application levels. In this section, the tools used in Electron for testing is discussed,

  - *Framework Testing* in Electron is a continuous testing framework to maintain stable builds[16]. Electron uses Jenkins for continuous integration tests on builds in Mac and Linux, AppVeyor[16] for Windows, and Circle CI for pull requests.

  - *Application Testing* - Spectron is a testing framework built upon WebdriverIO with helpers to access Electron APIs in tests and bundles ChromeDriver. WebDriverJs and WebdriverIO provide Node packages for testing with web driver[18]. Devtron, is a Electron Dev Tool developed on the Chrome Developer Tools for developers to inspect, monitor, and debug their Electron apps [19]. Devtron can be used to visualize dependency graphs, inspect the events and event listeners registered on the Electron app, monitor the IPC messages between the main process and renderer process and check the application for consistent code styles or linting.

Codeline models

According to Rozanski and Woods[3], the codeline models describe the source code structure, release process, configuration management, build and testing approaches.

- **Source Code Structure:** The source code organization structure of Electron is compliant with Chromium's Multi-Process model [22]. For example, `atom/` (C++ source code) and `lib/` (Javascript source code) contains modules `browser/`, `renderer/` and `common/` which contain submodules for main process, renderer process and main and renderers (both) respectively. All the scripts are maintained `script` folder. These are examples of classic Chromium style coding. The `third_party` dependencies of Electron

such as Node and Chromium's `libchromiumcontent` are found under the `/vendor` sub-directory to prevent a naming conflict with Chromium's Source Tree [20]. The complete directory organization of Electron based on the docs [7] with some additions to match the current repository is made below:

```
Electron
├── atom/ - C++ source code. Called atom because Electron was called atom at first.
|   ├── app/ - System entry code.
|   ├── browser/ - The frontend including the main window, UI, and all of the
|   |   main process things. This talks to the renderer to manage web pages.
|   |   ├── ui/ - Implementation of UI for different platforms.
|   |   |   ├── cocoa/ - Cocoa specific source code.
|   |   |   ├── win/ - Windows GUI specific source code.
|   |   |   └── x/ - X11 specific source code.
|   |   ├── api/ - The implementation of the main process APIs.
|   |   ├── net/ - Network related code.
|   |   ├── mac/ - Mac specific Objective-C source code.
|   |   └── resources/ - Icons, platform-dependent files, etc.
|   ├── renderer/ - Code that runs in renderer process.
|   |   └── api/ - The implementation of renderer process APIs.
|   └── common/ - Code that used by both the main and renderer processes,
|       including some utility functions and code to integrate node's message
|       loop into Chromium's message loop.
|       └── api/ - The implementation of common APIs, and foundations of
|           Electron's built-in modules.
├── brightray/ - Thin shim over libcc that makes it easier to use.
├── chromium_src/ - Source code copied from Chromium.
├── default_app/ - The default page to show when Electron is started without an app.
├── docs/ - Documentations.
├── lib/ - JavaScript source code.
|   ├── browser/ - Javascript main process initialization code.
|   |   └── api/ - Javascript API implementation.
|   ├── common/ - JavaScript used by both the main and renderer processes
|   |   └── api/ - Javascript API implementation.
|   └── renderer/ - Javascript renderer process initialization code.
|       └── api/ - Javascript API implementation.
├── script/ - scripts used for development purpose like building, packaging, testing, etc.
├── spec/ - Automatic tests.
├── tools/ - helper scripts used by gyp files, unlike script,
|           scripts put here should never be invoked by users directly.
├── vendor/ - source code for third-party dependencies.
├── electron.gyp - Building rules of Electron.
└── common.gypi - Compiler specific settings and building rules for other
  components like `node` and `breakpad`.
```

- **Build Approach:** Electron uses gyp for project generation which is built using ninja-build. The binaries of third-party frameworks that are used in Electron but are not supported in `gyp` build are found in the `external_binaries` directory.
- **Release Process:** The release process begins with the selection of the Release Candidate (RC) branch. In the RC branch the `prepare-release` script is run to perform checks, update version number and auto-generate draft release notes. After the `prepare-release` script execution is done, the release notes are compiled with major and minor changes done on the code base. The release takes place after the completion of the release draft and the release branch is pushed/ published.[24]
- **Configuration Management:** Electron is managed on Github and new versions of Electron are released through Github releases. Github provides a powerful version control system to maintain the source code which supports repeatability via branches and commits. The technical integrity of the main code is preserved as the master branch is always **version-less**. The configuration structures used are repositories, branches, labels, tags, milestones, issue trackers and pull requests.

3.1.3 Deployment View

In the previous sections, a view of the dependencies which make Electron work is described. This section summarizes the system requirements and additional frameworks to successfully run Electron. We now explain the Figure deployment view of Electron as illustrated in figure 7.

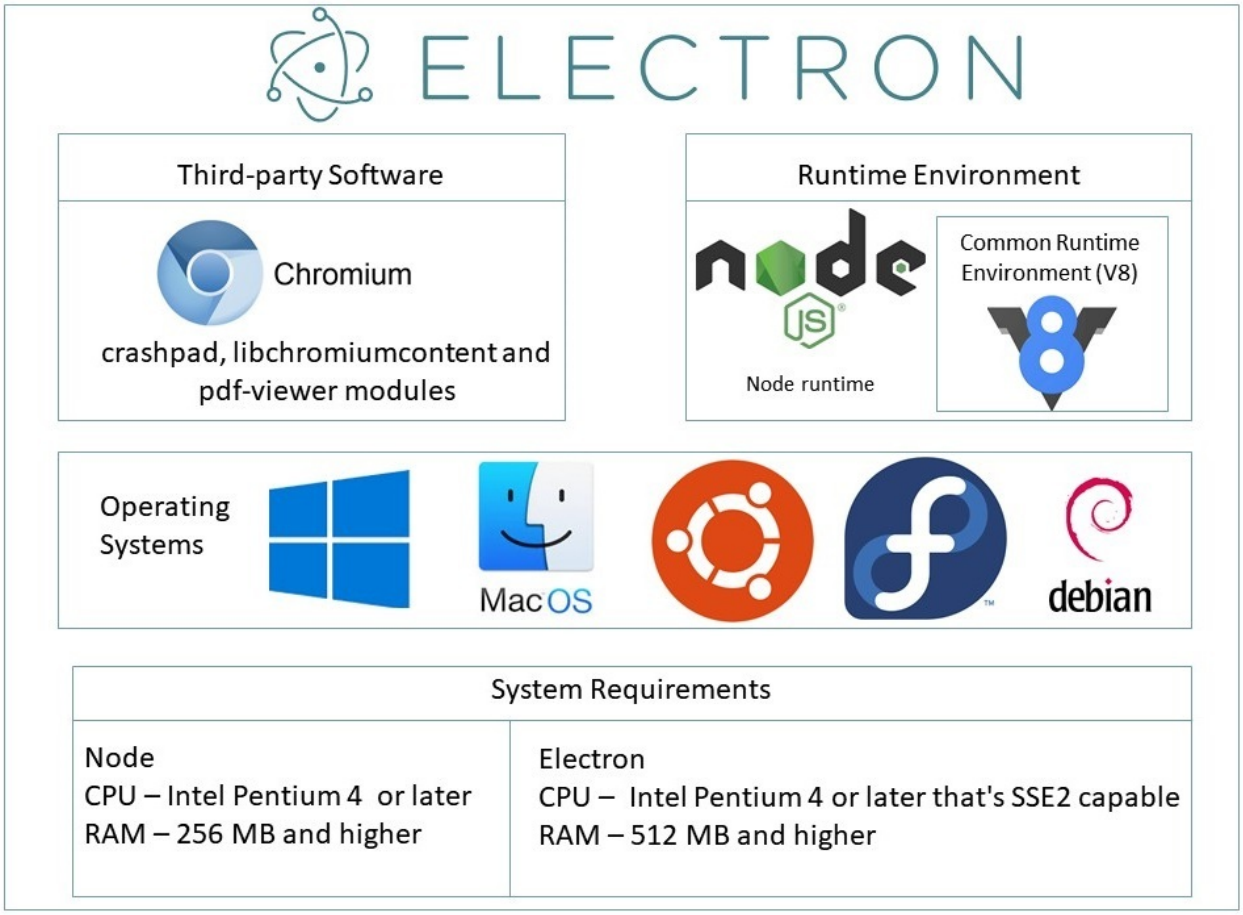


Figure 7: Deployment view of Electron.

- *Third-party software requirements*: Electron uses different Chromium modules and Node packages, like `libchromiumcontent` for content rendering, `pdf-viewer` for rendering `webui` pages, like `node-minidump` for processing minidump files, and other packages like `Chromium's Crashpad` for message logging. There is also a collection of binary frameworks that are bundled by Electron for making it truly cross-platform. These assets are downloaded as part of Electron's bootstrapping process and form the third-party system requirements for running Electron apps.
  - DirectX SDK.aspx)
  - MS Visual Studio C++ Runtime
  - Mantle
  - Reactive Cocoa
  - Squirrel for Mac
- *Runtime Environment*: Electron uses the Node runtime as its back-end component and this interacts with native Operating System. Electron and its two dependencies also use Google's V8 high-performance JavaScript engine which adds up as a runtime environment. Electron does not require additional downloads for runtime environments as they are embedded as pre-configured forks ready for use out-of-the-box.
- *Operating Systems*: Electron is built on Node and Chromium which are cross-platform which makes Electron and applications built-on it cross-platform. Thus, Electron is supported on Windows (both `32` and `64 bit` versions from Windows 7 and above), macOS(`64 bit` of macOS 10.9) and Linux (`arm` , `32` and `64 bit` versions of Ubuntu 12.04 and later, also works on Fedora 21 and Debian 8)

3.2 Architectural Perspectives

We give an overview of evolution perspective of Electron and also depict some interesting facts about the analyzed *technical debt* related to it. We then describe the security perspective of the Electron.

3.2.1 The Evolution Perspective and the related Technical Debt

Electron started as a fork of NW.js[6] for building GitHub's Atom editor and was called Atom-Shell in 2013. The project was developed a Cheng Zhao, an intern at GitHub. By spring 2014[2], Atom and Atom-Shell branched out into two separate projects. Atom-Shell was renamed as Electron in 2015 and in 2016 Electron reached version 1.0 with the support to publish apps to Mac and Windows App Store. In the perspecitve of technical debt(TD), this is an indication of single-point of failure in terms of only one contributor Cheng Zhao, which can be inferred from the code additions to the project in Figure 8.

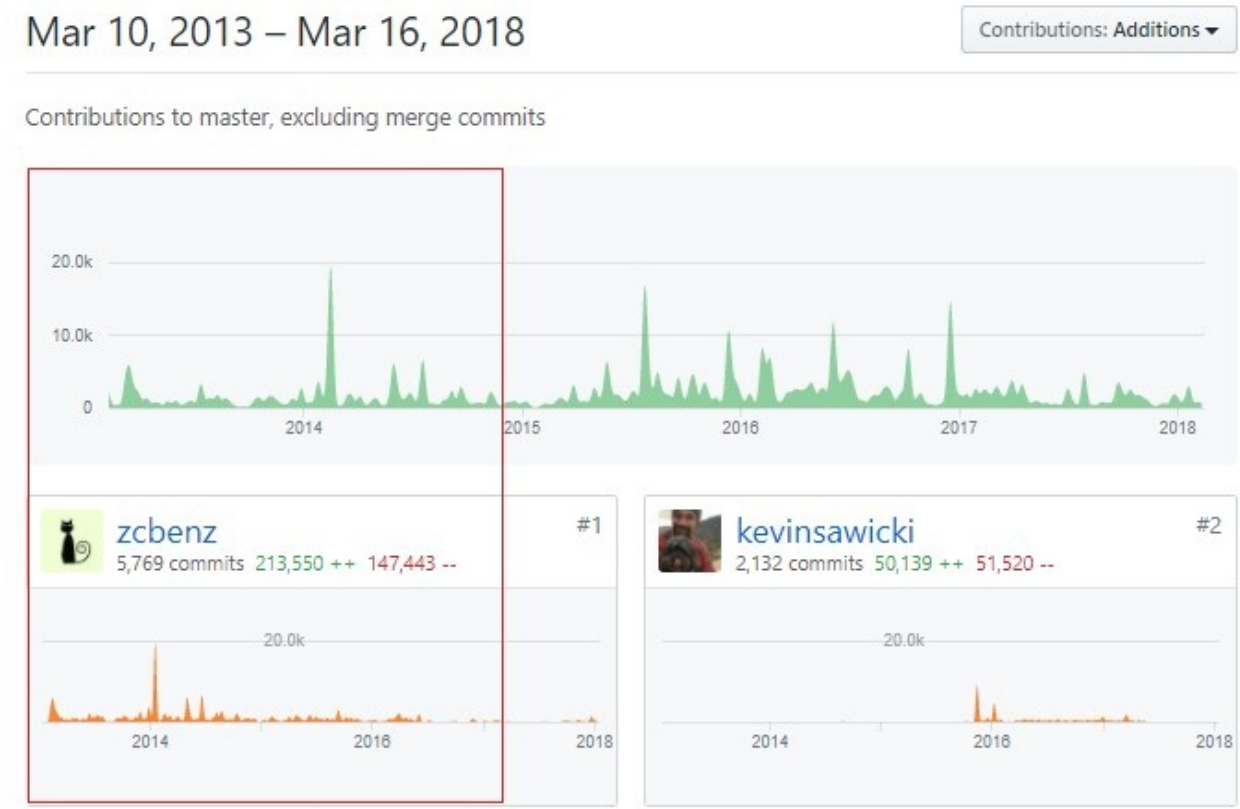


Figure 8: Electron's dependence on Cheng Zhao, a single-point of failure type of Technical Debt

After 2015, around the time when Electron branched out from Atom more core developers like @zeke joined and the single-point of failure TD started to reduce gradually as seen in Figure 9. To minimize TD in the code level the core developers standardized writing and testing contributions to ensure stable builds which has helped Electron develop at a rapid pace attracting many new contributors to the project.

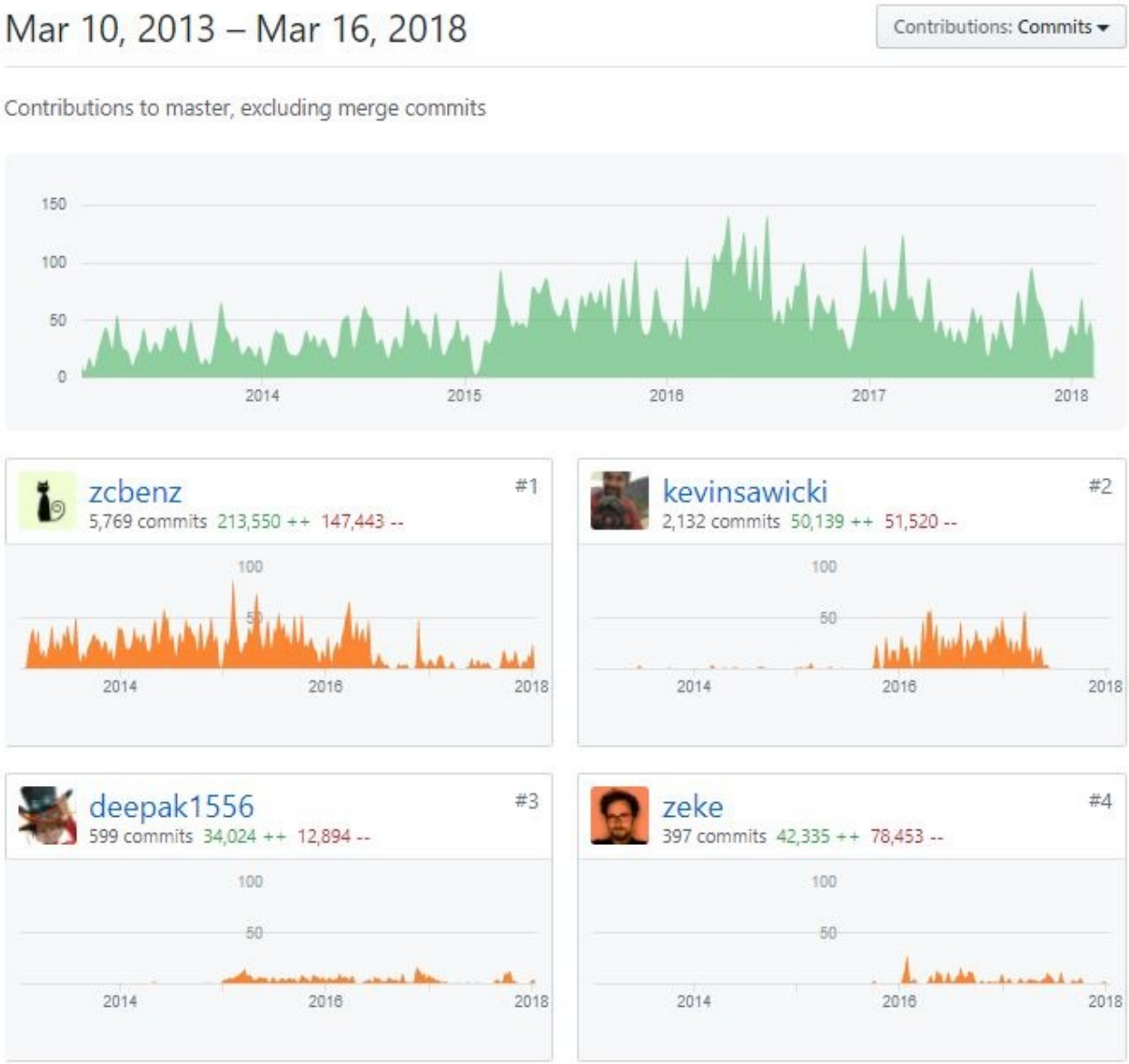


Figure 9: Electron's development timeline

From Rozanski and Woods 3, the evolution perspective is defined by:

- **Magnitude and Speed of Change:** At the time of testing, the stable version was `1.8.4`. Given the speed of evolution for Electron and still staying at the major version `1.x.x` from 2016 till Spring 2018 indicates stability of the platform. This indicates that the magnitude of major change in Electron is long-term and minor releases are short-term in nature.
- **Product Management/ Integrators** are the architects of the Electron. The core developers at Github like @zcbenz,@zeke and @ckerr are the integrators. Their challenge is to keep Electron releases stable as only pull requests (PR) that pass all the continuous integration checks and review requirements are merged. The code reviews are done by the integrators and there is a thorough discussion on the proposed change to make sure that the integrator and the contributor are on the same page, and this results in maintaining quality across contributions. PRs are subject to CI checks on CircleCI, Appveyor, Travis and Jenkins for different OS so that the changes are non-breaking, forward and backward compatible across all Electron versions and satisfy the contributing guidelines like linting of code, etc.
- **Dimension of Change:** Platform evolution are the major version updates like `v2.0.0` in which even the versioning strategy has changed to `semver`. This requires Electron to change for every major update of its dependencies (Node and Chromium). The functional evolution is periodic through patch and minor version increments.
- **Changes driven by External Factors:** Electron evolves with the version of Chromium which is used for rendering the GUI components. Since, Electron uses custom embeddings of its dependencies the Chromium update could be done at a later time and need not be an immediate one. But when the Electron 2.0 arrives, Electron will conform to `semver` which requires Electron to

change for every major update of Node and Chromium.

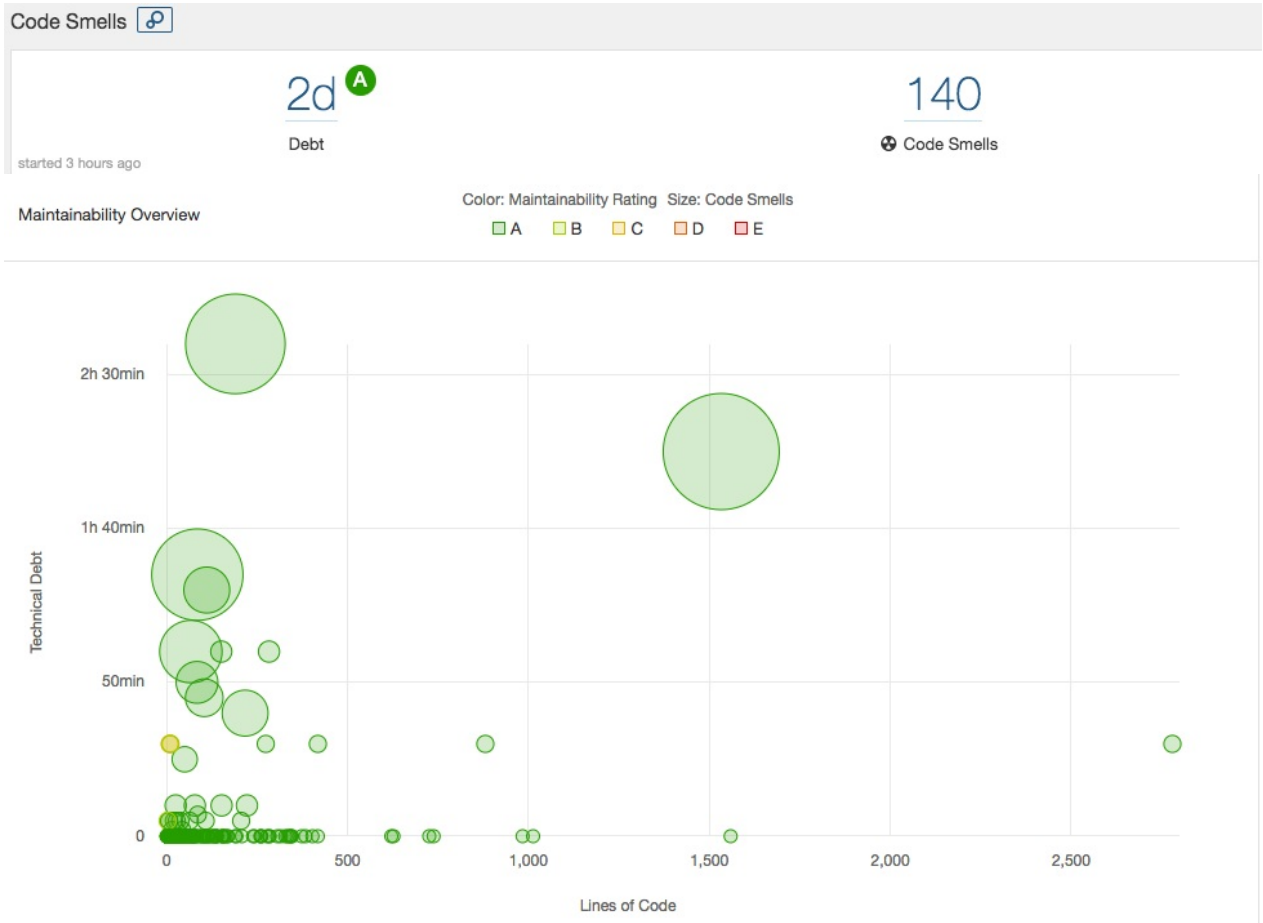
At the time of writing this chapter the major update version 2.0.0 is in the final stages of development and we analyzed the technical debt on the versionless `master` during an impending new version release. The tools used were:

Name of Tool	Purpose
SonarQube	Code smells, code duplicates, finding bugs and security vulnerability
github-grep	for finding `FIXME` and `TODO`
Github Insights	to find indicators of technical debt from contributions and contributors

Results from SonarQube

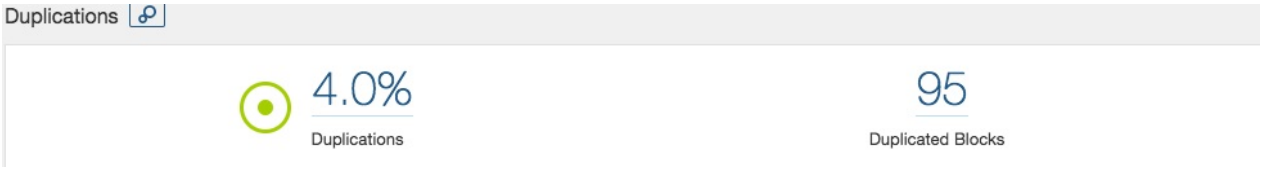
Following are the results we got from using SonarQube to assess the code quality of core electron repository:

- Code Smells: Code smell analysis shows 2 days technical debt, which is very low considering the size and lines of codes in Electron. The 140 code smells depicted were not representative of real issues and were from the limitation of selecting primary language in SonarQube as JavaScript. About 70 of these were about refactoring function names which is not applicable to C++ files. The real code smell founds were mostly in testing and scripting files which don't form an important component for core of Electron and does not affect it's stability. The figure also shows the maintainability rating `A` for all files except one testing file. Hence, we can say current version of `master` of electron is maintained pretty well.

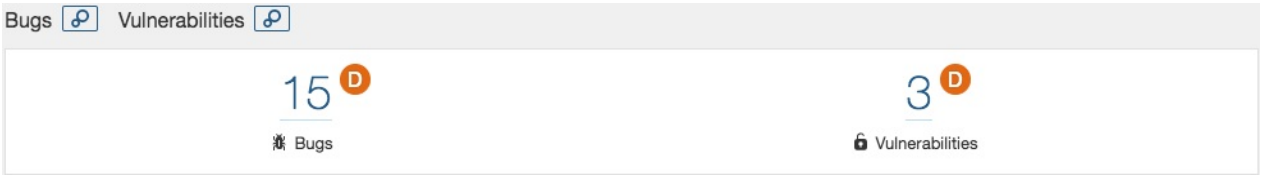


- Duplicates: There are only 4% duplications found which again comes from testing code. Our analysis states that these duplicates cannot be avoided due to the nature of `describe-it` statements in Mocha and Chai testing frameworks.

Still, one can say that testing methods have a little bit of technical debt in terms of code smells and duplicates but electron developers do discuss about these through TODOs and FIXMEs as explained in point 4 below.



- Bugs and Vulnerabilities: SonarQube detected 15 bugs and 3 security vulnerabilities. The bugs were related to coding style such as defining functions outside loop and argument mismatch. Only one of them was critical in nature and was related to how `eval` function was used at runtime to call an object's property. The solution is to call the object's property at compile time. It also estimates security remediation would take only about an hour to fix all three of them. Again, this is extremely low number of bugs and security warnings compares to the size of electron.



Results from github-grep for TODOs and FIXMEs

The developers of Electron do discuss about the technical debt there is and they communicate through code as well as github issue tracking by mentioning TODOs and FIXMEs. We used `github -grep` to find TODOs and FIXMEs in the code. In total there were 79 TODOs and 46 FIXMEs found. Each of them decribed who wrote it and/or who will fix it. Examples are shown below:

- `atom/browser/api/atom_api_app.cc: // TODO(juturu): Remove in 2.0, deprecate before release`
- `chromium_src/chrome/browser/process_singleton.h: // TODO(brettw): Make the implementation of this method non-platform-specific`
- `spec/api-crash-reporter-spec.js: // TODO(alexeykuzmin): Skip the test instead of marking it as passed.`
- `spec/webview-spec.js: // FIXME(alexeykuzmin): Skip the test.`
- `atom/browser/web_contents_preferences.h: // FIXME(zcbenz): This method does not belong here.`

Majority of them were about skipping tests instead of marking them passed in the version 2.0. However, by doing this we got to know the technical debt (still relatively very small compared to repo's size) not found by automatic finders like SonarQube and can only be found through developer insights and project's long term plans. The TODOs in core Electron's code represent deprecations to be done in release 2.0 of the release. Our solution to this is that these 2.0 TODOs should not be in the versionless master branch but 1.8.4 (current production version's) branch.

There is also discussion about these on github issue tracking. The TODOs and FIXMEs there are on much higher abstraction level as described in this recent example on:

- https://github.com/electron/electron/issues/10836: List of hacks and workarounds made during the upgrade to Chromium 61. They are to be fixed before Chromium 61 is merged or right after. Most of the issues are fixed already. This reinforces that Electron's developers are very active and keep it's technical debt low.
- https://github.com/electron/electron/issues/11242: This described the debt in terms of updating documentation (Remove async menu from the docs) that came with upgrades in Chromium 61.

We can infer from this activity that Electron developer's community is well aware of the technical debt that is arising from the new incoming version 2.0 and is actively working towards reducing the same. This makes evolution of Electron future-proof and their steps towards `semver`, where they will update Electron with every major update of its dependencies reinforces this fact.

3.2.2 Security Perspective

It is important to distinguish that Electron is not a web-browser and the applications are built using Chromium as a renderer and Node to make remote calls to sources on the web. The security of applications built on Electron depends on the current, latest version of Chromium. Chromium has an off-the-shelf sandbox to run processes which can freely use CPU and memory. However, being a restrictive environment, there are well-defined policies for processes which prevents bugs and attacks during IO operations.

To provide applications with desktop functionality, the Electron team modified Chromium to introduce a runtime to access native APIs as well as Node.js's built in and third-party modules. To do this the Chromium sandbox protection was disabled, meaning any application running inside Electron is given unfiltered access to the operating system.

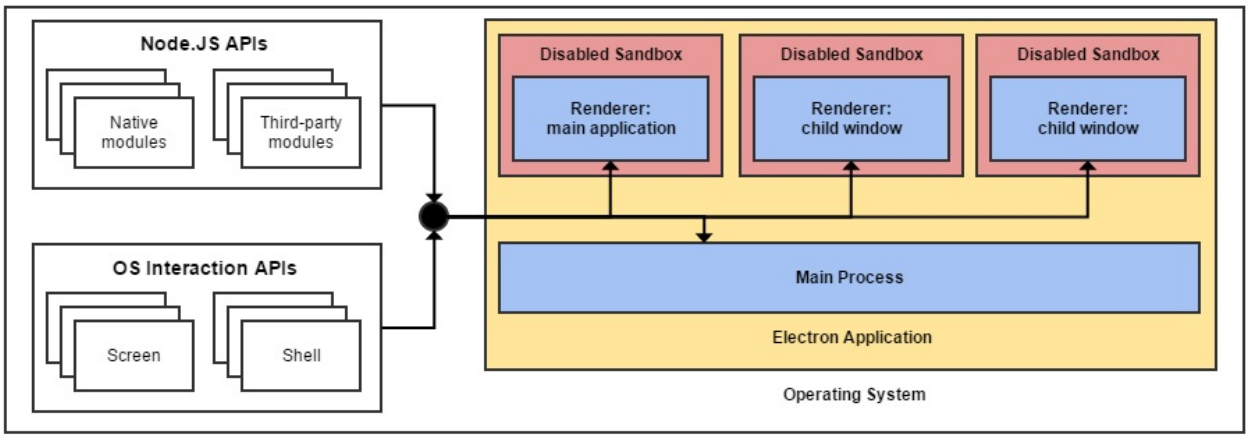


Figure 15: Electron's architecture combined with disabled Chromium sandboxes [28]

This is not a concern for a lot of Electron applications, especially those that don't pull in remote data and need extensive operating system privileges anyway (such as an IDE). However, this is a (potentially) serious and unnecessary security risk for applications that do pull in remote data, any successful XSS attack would give the attacker full control over the victim's machine! It's worth stressing that a successful XSS attack within a sandboxed application is still a catastrophic event that can do untold damage. Even restricted to the sandbox, an attack can execute scripts to hijack user sessions, deface web sites, insert hostile content, redirect users and install malware. Nonetheless, unless they escape the sandbox the potential for damage stops at the browser level with the operating system left unharmed and only data explicitly shared with the browser at risk.[28]

A cross-site-scripting (XSS) attack is more dangerous if an attacker can jump out of the renderer process and execute code on the user's computer. Cross-site-scripting attacks are fairly common - and while an issue, their power is usually limited to messing with the website that they are executed on. Disabling Node.js integration helps prevent an XSS from being escalated into a so-called "Remote Code Execution" (RCE) attack.[29]

It's worth pointing out that there's an open discussion around Electron's security model on GitHub, so further changes and improvements are likely to be made once this discussion is finalised.

To summarise, Electron currently has its fair share of security issues that complicate the process of building secure applications. Various workarounds also exist that circumvent turning node integration off in the event of an XSS attack. One major goal should be a more secure way for applications that do need some node and desktop integration to interact with the operating system. Fortunately, due to the framework's popularity and active community the Electron team are already making good progress to resolving these issues.[28]

4. Conclusions

We can summarize our detailed analysis through the following points:

- Electron makes thing easy by providing a framework for developing cross-platform desktop apps through web technologies.
- Electron has a large and friendly community, who is always eager to welcome new contributors. The experience is based on our own contributions. One of the core developers @Zeke even did a video call to help us with coding issues for the contribution.
- Lots of applications are dependent on Electron and hence, extra focus is given on Electron's stability at any given point of time.
- Electron is backed up by Github and other large communities. The developers of Electron are well-aware of the technical debt that comes with evolving and actively discuss and contribute towards keeping it low.
- Electron is gaining a lot of attention and the contributor community is large, skilled and fast growing, and so when you might be reading this chapter, a new version 2.x.x must be available.
- In 2.x.x, Electron has promised to update according to every major update to its depedencies making it more secure and robust.

Do you want to contribute to Electron as well? Take a look at Electron and the contributing guidelines for the Electron project. It's our guarantee that you will be welcomed with open arms by team Electron and will provide help and feedback by all means possible for your contributions.

References

1. Electron (software framework)(2018). [online] Available at: https://en.wikipedia.org/wiki/Electron_(software_framework) [Accessed 23 Feb. 2018].

2. Steve Kinney (2018), Introducing Electron in Electron in Action(pp. 1-15), Manning Publications.

3. Rozanski, N., & Woods, E. (2012). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.

4. Electron: Tutorial (2018). [online] Available at: https://electronjs.org/docs/tutorial/ [Accessed 23 Feb. 2018].

5. Olander, S., & Landin, A. (2005). Evaluation of stakeholder influence in the implementation of construction projects. International journal of project management, 23(4), 321-328.

6. Lynch, A. (2017). Beyond The Browser: From Web Apps To Desktop Apps. [online] Available at: https://www.smashingmagazine.com/2017/03/beyond-browser-web-desktop-apps/ [Accessed at 4 March, 2018]

7. Electron: Source code structure(2018). [online] Available at: https://github.com/electron/electron/edit/master/docs/development/source-code-directory-structure.md [Accessed 4 March, 2018]

8. Betts, P. (2016). Building Hybrid Applications with Electron. [online] Available at: https://slack.engineering/building-hybrid-applications-with-electron-dc67686de5fb [Accessed at 4 March, 2018]

9. Betts, P. (2016). Using ES2015 with Electron—introducing electron-compile. [online] Available at: https://slack.engineering/using-es2015-with-electron-introducing-electron-compile-2a0e5ccbadb6 [Accessed at 4 March, 2018]

10. Electron's dependecies (2018), [online] Available at: https://david-dm.org/electron/electron?type=dev [Accessed at 4 March, 2018]

11. Elctron API documentation, [online] Available at: from https://electronjs.org/docs/api [Accessed at 4 March, 2018]

12. Nokes, C. (2016). Deep dive into Electron's main and renderer processes.[online] Available at: https://codeburst.io/deep-dive-into-electrons-main-and-renderer-processes-7a9599d5c9e2 [Accessed at 5 March, 2018]

13. Norris, T. (2015). Understanding the Nodejs Event Loop. [online] Available at: http://nodesource.com/blog/understanding-the-nodejs-event-loop/ [Accessed at 5 March, 2018]

14. Crashpad README(2018). [online] Available at: https://github.com/electron/crashpad/blob/master/doc/overview_design.md [Accessed at 5 March, 2018]

15. Crashpad- Development Notes (2018). [online] Available at: https://github.com/electron/crashpad/blob/master/doc/developing.md [Accessed at 5 March, 2018]

16. Electronjs.org Testing on Headless CI (2018). [online] Available at: https://electronjs.org/docs/tutorial/testing-on-headless-ci [Accessed at 5 March, 2018]

17. Electronjs.org Testing (2018). [online] Available at: https://electronjs.org/docs/development/testing [Accessed at 5 March, 2018]

18. Spectron. (2018). [online] Available at: https://github.com/electron/spectron [Accessed at 5 March, 2018]

19. Devtron. (2018). [online] Available at: https://github.com/electron/devtron [Accessed at 5 March, 2018]

20. Electronjs.org source code strucutre. (2018). Source Code Directory Structure. [online] Available at: https://electronjs.org/docs/development/source-code-directory-structure [Accessed at 6 March, 2018]

21. Electronjs.org/ build overview. (2018). Build System Overview. [online] Available at: https://electronjs.org/docs/development/build-system-overview [Accessed at 6 March, 2018]

22. Chromium.org/ multi-process architecture. (2018). Multi-Process Architecture. [online] Available at: https://www.chromium.org/developers/design-documents/multi-process-architecture [Accessed at 6 March, 2018]

23. Lord, J. (2018). Jlordus. [online] Available at: http://jlord.us/essential-electron/ [Accessed at 6 March, 2018]

24. Electronjs.org Releasing (2018). [online] Available at: https://electronjs.org/docs/development/releasing [Accessed at 6 March, 2018]

25. Electronjs.org. (2018). Releasing. [online] Available at: https://electronjs.org/docs/tutorial/application-packaging [Accessed at 6 March, 2018]

26. Cunningham, W. (2011). Ward Explains Debt Metaphor. [online] Available at: http://c2.com/cgi/wiki?WardExplainsDebtMetaphor [Accessed at 15 March, 2018]

27. Electron's Versioning Documents. [online] Available at: https://github.com/electron/electron/blob/master/docs/tutorial/electron-versioning.md [Accessed at 15 March, 2018]

28. Kerr, D. (2018). As It Stands - Electron Security. [online] Available at: http://blog.scottlogic.com/2016/03/09/As-It-Stands-Electron-Security.html [Accessed 1 Apr. 2018].

29. Electronjs.org. (2018). Security. [online] Available at: https://electronjs.org/docs/tutorial/security [Accessed 1 Apr. 2018].

Godot

By Alkis Antoniades, Miriam Doorn, Marie Kegeleers, Felix Yang

Delft University of Technology, 2018

Introduction

Godot is a community-developed, cross-platform, open-source game engine released under the MIT license and is free for commercial applications. A large and growing online community actively contributes to the development by coding, testing, documenting and promoting the Godot Project.

In this chapter, we will describe our research, analysis and documentation of the architecture and development of the Godot Engine. Who are the stakeholders most interested in the development? What are the qualities which are most important for such a system, and how do we ensure those qualities are implemented in the system?

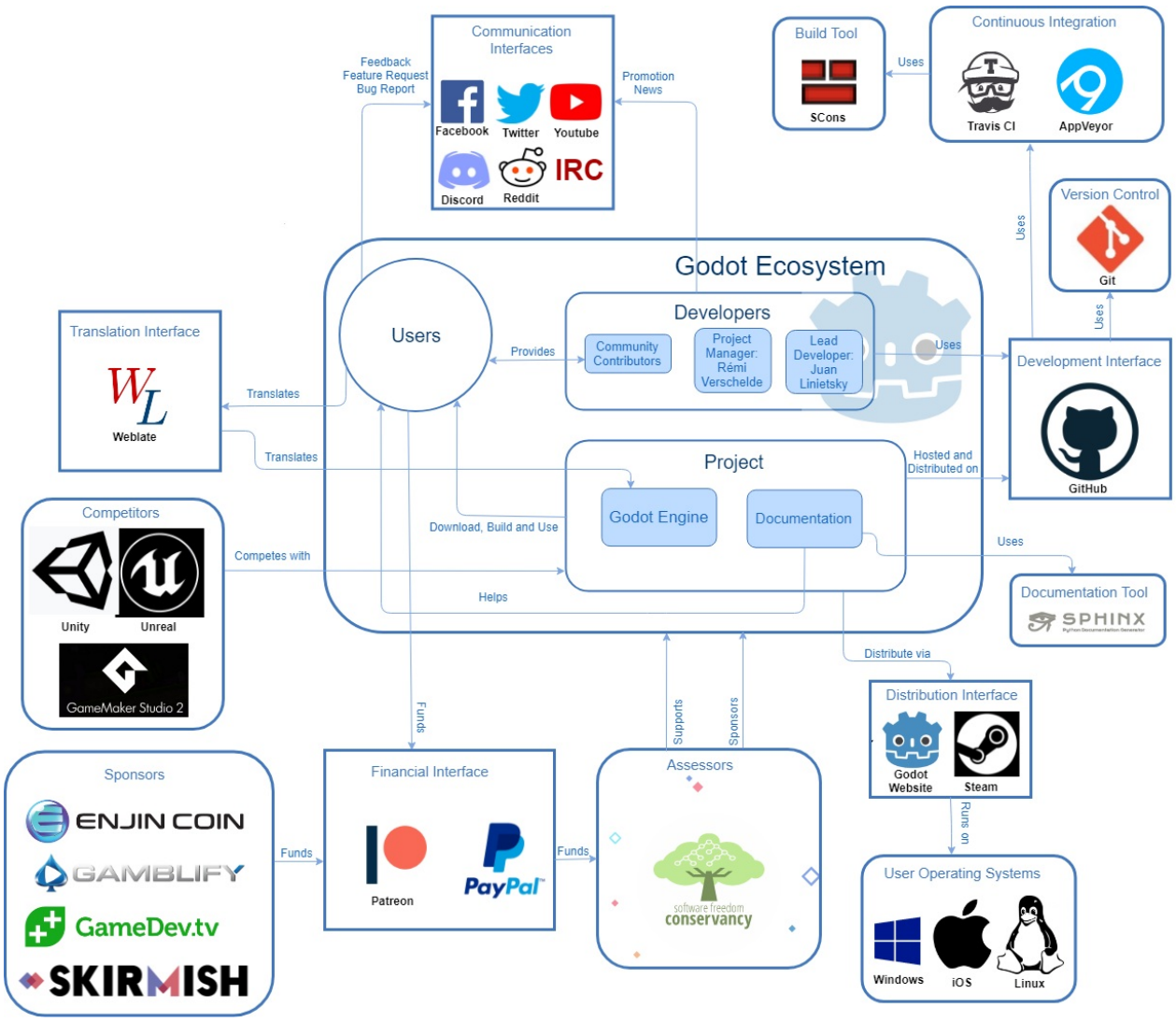
Table of contents

Stakeholder Analysis

Stakeholder analysis helps identify responsibilities of persons involved in the project. The following table shows the types of stakeholders present in the project and their involvement. Due to the project being completely non-profit, community-developed and released under MIT license, the stakeholder types are quite unique.

Stakeholder	Person/Entity	Involvement
Lead Developer	Juan Linietsky (**reduz**)	Creation of most of the code, and high-level decision making.
Committee Members	Juan Linietsky (**reduz**), Ariel Manzur (**punto**), Rémi Verschelde (**akien-mga**), George Marques (**vnen**), and Andreas Haas (**Hinsbart**)	Financial management.
Project Manager	Rémi Verschelde (**akien-mga**)	Maintainer, integrator and manager of Godot repositories, also acts as the representative of Godot Engine to the public.
Senior Developers	Members of Godot Engine	Assist the lead developer with software development, and assist the project manager with issues & pull requests management and integration.
Community Contributors	Contributors of any repositories of Godot Engine	Bugfixing, the creation of minor new features, documentation creation and maintenance, testing, etc.
Users	Individuals or companies that develop games with the Godot Engine	Use the product, report encountered bugs, and request new features.
Sponsors	Patrons of the Godot Project on Patreon] (https://www.patreon.com/godotengine)	Support the development with money, and lean in the development process with their opinions.
Competitors	Unity 3D, Unreal, etc	Game engines offering similar features and tools.

Context Viewpoint



System Scope and Responsibilities

Godot is a game engine that provides a large set of tools used in game development, including an editor with a graphical interface, which runs on Windows, macOS, Linux and BSD, and can create games targeting PC, console, mobile, and web platforms. The project extends to include documentation on how to use the editor, as well as translations for it.

System Responsibilities:

- Support for development of both 2D and 3D games.
- Providing an intuitive, cohesive and integrated development environment.
- Cross-platform support for development environment (Windows, Mac OS, Linux).
- Even greater cross-platform support for publishing games (mobile, web, console, PC).
- Performance should be comparable to mainstream engines such as Unity 3D and Unreal.
- Clear and comprehensive documentation.
- Free for all applications.
- Open to customization.

Entities and Interfaces of Interests

Funding

The Godot project is being crowdfunded on Patreon and by direct donations using Paypal. At the moment there are four corporate sponsors through Patreon, with the rest of the donors being individual members of the Godot community. The Software Freedom Conservancy (SFC) acts as a fiscal sponsor, and any funding goes through it.

Development

Juan Linietsky and Rémi Verschelde manage a large group of community developers who contribute to the Godot Engine code base via GitHub. GitHub is used as an interface where developers and users can access the codebase that is hosted there, and is also used for discussions and project management.

When a pull request is made for a code contribution on GitHub, Travis CI and AppVeyor are deployed to check whether the changes suggested pass certain tests. SCons is used as a tool to generate builds for multiple platforms.

Documentation

The source code for the documentation of the Godot Engine has its own github repo, and contributors are encouraged to make additions. The documentation is accessible on the Godot Docs website, as well as directly inside the Godot Editor.

Translation

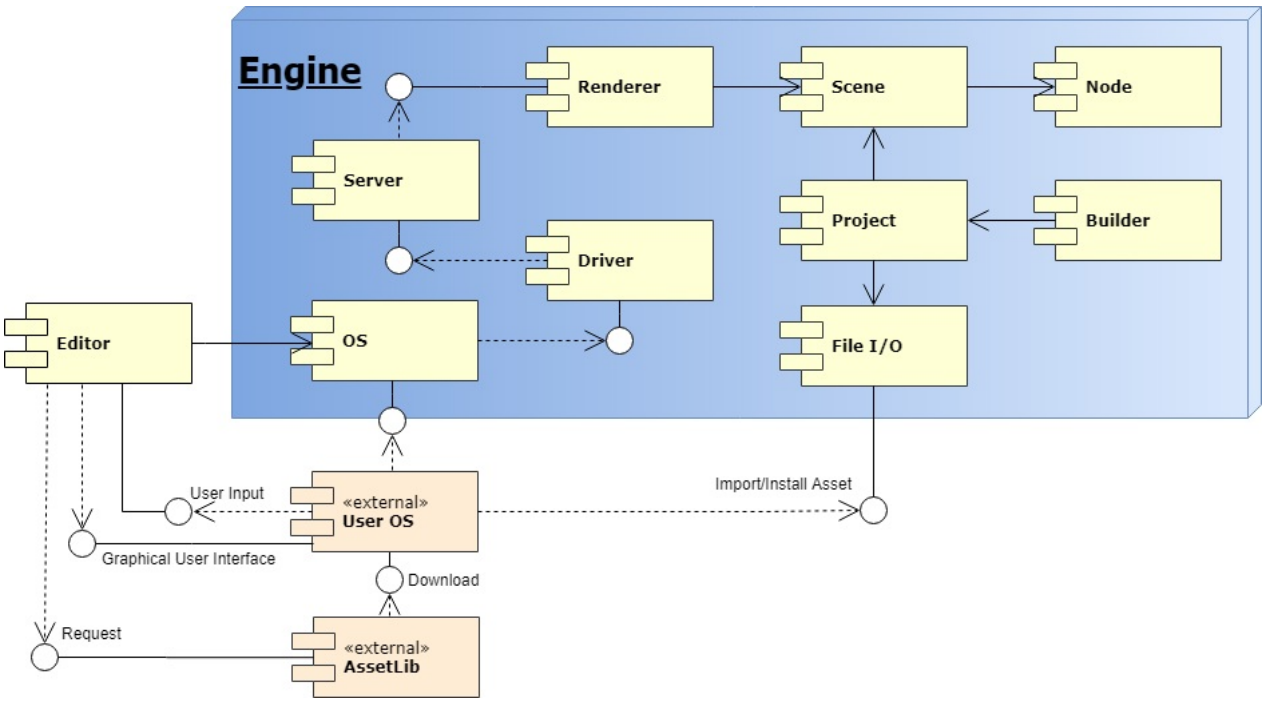
The translation interface will be described in detail in the section on Internationalization.

Communication

The community of the project uses many channels to interact, such as Twitter, Reddit, Facebook, Discord, IRC, etc, where members help each other when they have questions or want to discuss the development of games, as well as the development of the Godot engine project. Users can act as advocates for the Godot engine by showcasing the games they make using the Godot editor.

When a stable release is ready for deployment, it gets released on the Godot official website and on Steam for supported platforms.

Functional Viewpoint



The diagram above illustrates the functional structure of the Godot Engine, which consists of multiple internal and external entities, and the interfaces and dependencies between them.

Engine

Most internal entities are contained within the group *Engine*, the core of the whole system. *Engine* renders game scenes and executes game logic.

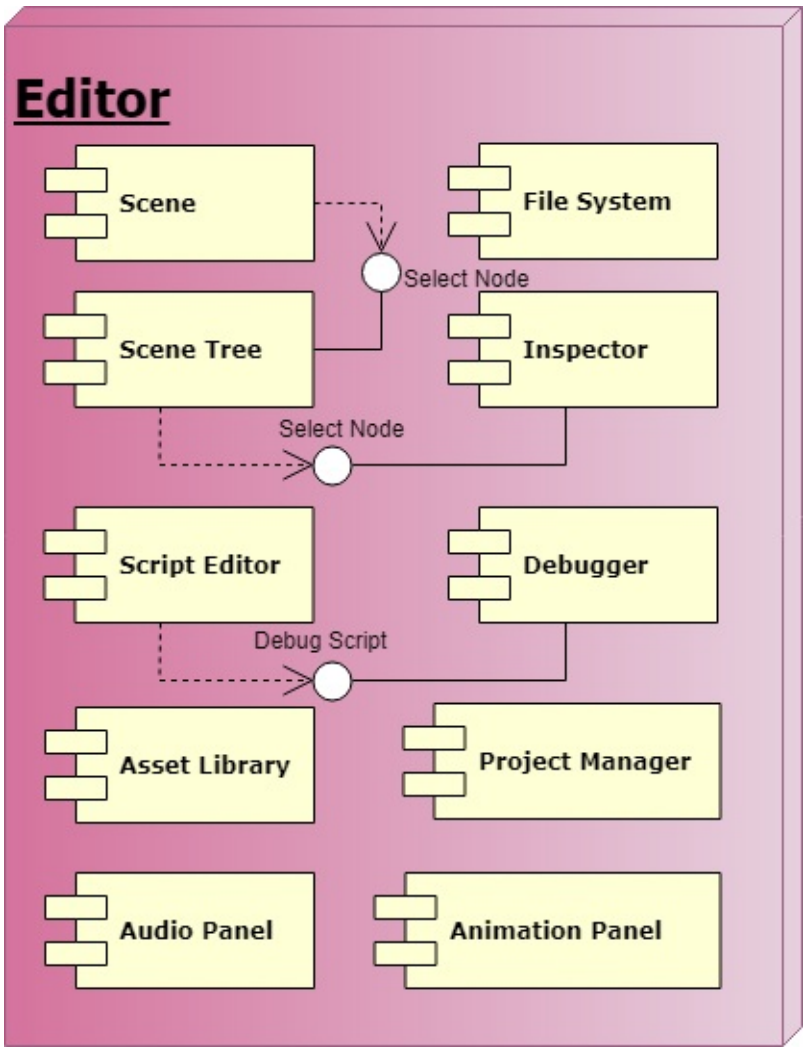
The biggest entity within *Engine* is *OS*. It serves many fundamental purposes, including communicating with *User OS*, initializing the system, etc.

Driver and *Server* together implement graphics and audio effects in games, with the difference being that the former handles low-level communication with APIs and is hidden from the user, while the latter is visible to script writers.

Renderer makes use of *Server* and renders the current scene, which makes it dependent on the *Scene* entity. Scenes in the Godot engine consist of multiple nodes in a tree structure, which makes *Scene* dependent on *Node*.

A project in the Godot engine consists of multiple scenes, which explains the dependency from *Project* to *Scene*. A project is also a basic unit the game builder takes in to build game executables, thus the dependency from *Builder* to *Project*. Any changes to a project that need to be read or written are handled via *File I/O*.

Editor

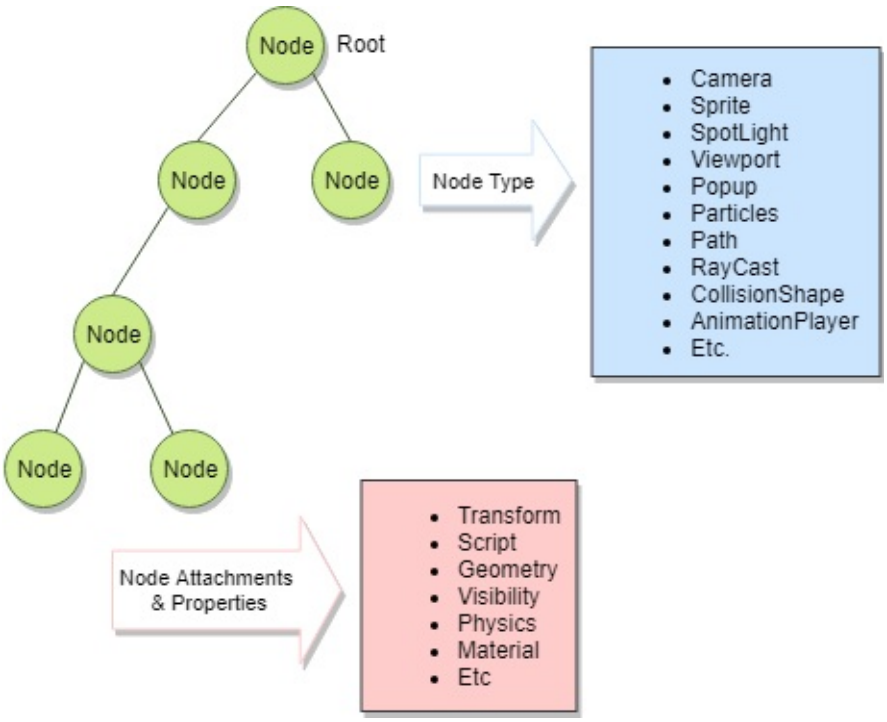


Editor serves as an interface between *User OS* and *Engine OS*, to help users manage projects easier, and display information from *Engine* in an intuitive way. The above diagram shows the many sub-entities of *Engine* that offer functionality to the user in a Graphical User Interface (GUI).

Project Manager is used to create new projects, load existing ones, and it contains a list of templates, which are essentially demos showcasing different game mechanics that developers can use to simplify certain aspects of their project development.

File System facilitates the import of game assets such as textures, 3D models, etc, and the creation of scripts which are used for specifying game logic, or shaders that can be attached to game objects, as well as game scenes which can consist of game assets with their attached scripts.

Scripts can be edited in the built-in *Script Editor*, which has its own *Debugger*.



Scene Tree can be used to create Nodes and edit the scene tree in a visual way. The above diagram shows a visual representation of a scene tree and the types of Nodes it can contain. Properties can be set to Nodes in the *Inspector*. *Scene* shows the Nodes in a 2D or 3D environment.

In *Animation Panel*, users can create and edit animations for nodes in the scene tree, including those that have gameplay related values.

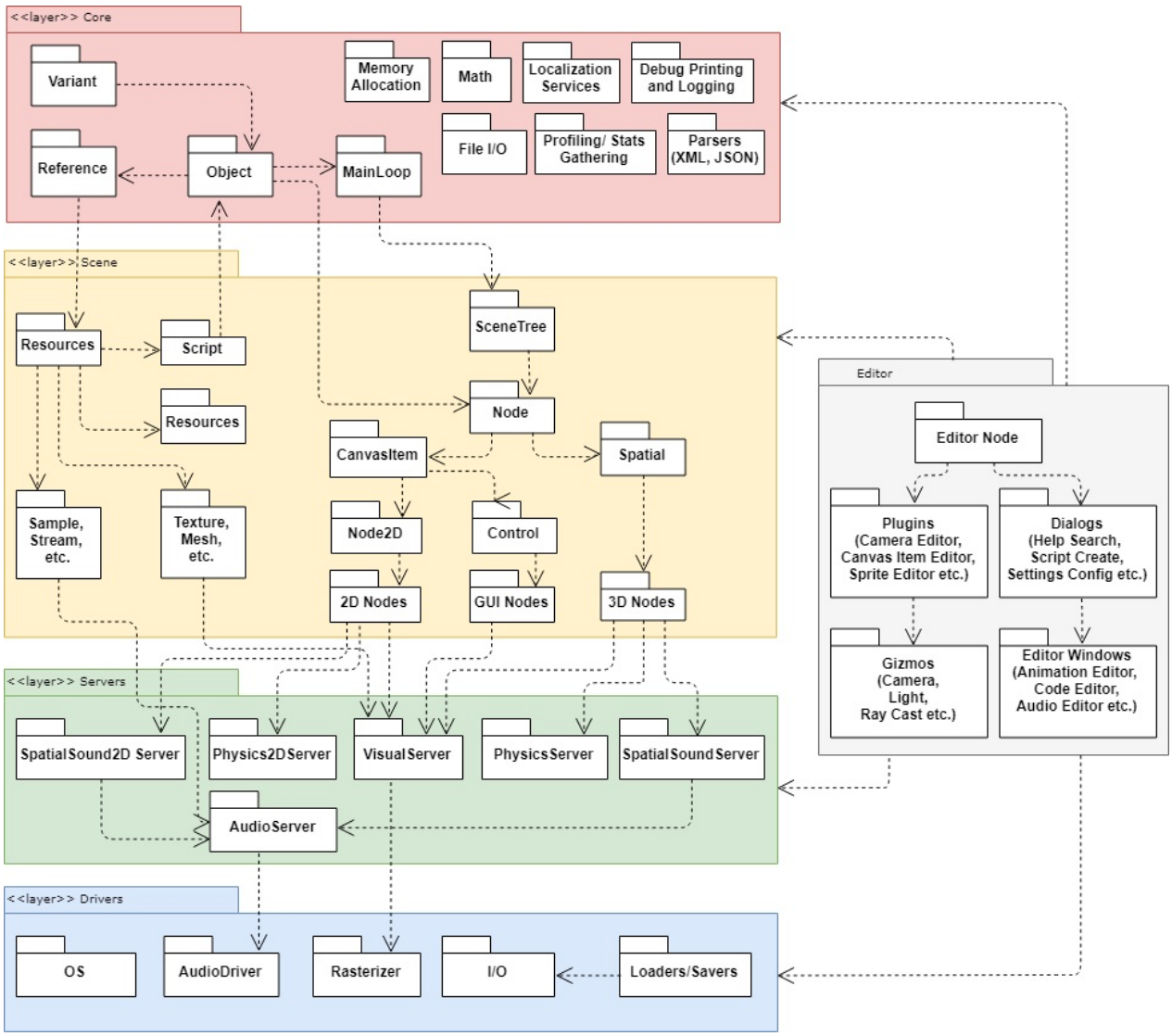
Audio Panel can be used to edit the settings of audio buses. Audio streams can be set to play from the target audio bus.

Through the *Asset Library* GUI, users can search for and download assets from *AssetLib*. The interface between the *Editor* and *AssetLib* server is based on the REST API

Development Viewpoint

Module Structure

The Godot Engine has a layered architecture with the modules organized in four layers with intermodule dependencies, something which the structure diagram below illustrates. This diagram is based on the architecture diagram found in the Godot Documentation, and even though this diagram was made over a year ago, the basic structure of the engine remains the same.



- Core: The core layer consists of the main loop of the program, which keeps it running, along with class modules that can be inherited from to form a consistent base for each class. The most important classes here are *Variant* and *Object*. *Variant* can contain all data types that are useful to the engine and can therefore be used as a dynamic type in C++. *Variants* can be set to *Objects* as properties. *Object* forms a general base for a class by providing elements like id, flags and get methods. *Reference* contains basic elements for classes that involve resources, configurations, parameters etc. All the nodes and resources inherit from the *Object* class. The core also contains many basic low level functionalities such as *Math* and *File Input/Output*.
- Scene: This layer contains all modules concerning the scene which is everything that the user interacts with when building a game. Each scene is built up using a *SceneTree* which consists of nodes, instances of classes that inherited from the *Node* class. Some examples of nodes are controls, 3D objects and GUI elements.
- Servers: This layer contains detailed implementation for aspects like Graphics, Physics, Audio etc, and users can interact with them via scripts and the editor. The most prevalent one is the *VisualServer*, since everything that is visible in-game gets processed by it.
- Drivers: This layer implements functionalities provided by the *Server* layout on a lower level. The drivers layer handles various platforms and exterior systems, thus making the low-level details opaque to the Servers layer. For example, the *I/O* and *Loaders/Savers* deal with different read and write methods on different platforms, etc..
- Editor: The editor group is a collection of all modules that make up the Godot Editor application. The editor is not part of the Godot Engine per se, and in fact it has been created using it, and is thereby dependent on modules from all the layers in it.

Common Design

Graphics Rendering

One of the concerns of the Godot Engine is that games made using the engine should be portable to different platforms and operating systems. You can imagine that rendering for a mobile game is done differently from rendering for a Linux desktop. To enable this quality, all classes that draw graphics have to interface with the Visual server singleton class, which then interfaces with the OpenGL ES API for rasterization and rendering.

Portability

All components, including custom user modules, must include an *SCsub* file, which contains Python code that enables a third-party software, SCons, to include them in the engine build. SCons is an open source cross-platform software construction tool that uses Python scripts for configuration files, and according to the Godot engine authors, it plays a pivotal role since it facilitates building the engine for different platforms, without breaking the build. Custom user modules also need to have a config.py file, which states whether it is ok to build for a specific platform.

Writing Scripts

GDScript is a high level, dynamically-typed programming language similar in syntax to Python, which has been developed to make it easy and intuitive for game developers to use the Godot Engine. For the scripting engine to determine the available types when coding in GDScript, *ClassDB* is used, which is a static class holding a list of registered classes that inherit from *Object*, along with dynamic bindings to all their method properties. Each class desired to be available in GDScript needs to be registered to *ClassDB*, and should users creating modules in C++ want the classes to be available in GDScript, they need to apply these bindings.

Game developers can also write code in VisualScript, which makes the coding process more visual to allow for a lower entry barrier to using the engine, and C#. C++ is the programming language used for developing the game engine itself.

Memory Usage

Godot employs various tools for tracking memory usage in a game, especially during debug, so the regular C and C++ library calls should be avoided in favor of Godot-provided macros, such as *memalloc()*, instead of *malloc*, etc, and for memory allocation in the style of C++, special macros are provided. Objects are also notified right after they are created, and right before they are deleted. The *DVector<>* template is also provided for dynamic memory purposes.

Use of third-party libraries

The Godot Engine makes use of third-party libraries that either have the same MIT-license as the Godot Engine or one that is similarly open and free, with the full list of libraries used being available in the documentation.

Standardization of Design and Testing

Standardization of Design and Code

Defining code style standards ensures good collaboration and overall quality. The Godot developers advice code contributors to conform to the existing style of code. Each pull request has a code review by one of the core developers, to ensure that the code is clean and maintainable. The code style rules are formalized in the .clang format file found in the Godot repository, and the style for C++ code is checked automatically using Travis CI when a contributor creates a pull request.

There are very few comments to be found in the code, and the core developers say this is mostly because people don't bother to make the effort. Having comments in non-trivial code would be a useful addition, however there are no guidelines defined for code commenting, because they believe this would disencourage people from contributing and it would add an extra step in code review to determine whether certain comments in the code are necessary.

For Python scripts, such as those used in the SCons build system, the PEP-8 style guide is adhered to. For Java scripting there is no chosen style standard, and contributors are encouraged to keep their code as clean as possible.

There are also guidelines on the proper format to use when filing an issue on the project's GitHub page.

The Godot documentation offers guidelines on organizing issues posted on Github, including a list of the labels currently defined in the Godot repository.

Test and Build Strategy

Before a pull request is merged, automated tests are run on the code submitted. For continuous integration, the Godot project uses Travis CI and Appveyor. Travis CI is used to make sure the code has the correct formatting, and Appveyor is used to make sure the project builds without problems. For creating builds, the tool SCons is used.

The engine is mostly tested by its users. There are reliable unofficial sites offering daily builds such as here and here, and quite a few users work with them, therefore when there are pressing issues they are found and fixed quickly, often within hours. Many things cannot be tested automatically, such as problems in the editor, something which requires users to notice the issue and report it to the developers. Furthermore, since the code base is changing so rapidly and features are being added constantly, it would be difficult to keep automated tests up-to-date, therefore making user testing the more agile approach.

Codeline Organization

Code Management

The Godot Project uses the "Pull Request workflow", which is common with projects on GitHub. Contributors fork the project, make modifications, and create a pull request, usually to the master branch. Other contributors can then review the code, comment on it and suggest changes when needed. When a pull request is approved, one of the core developers (mostly Rémi (**akien-mga**)) will merge it. You don't need to sign a CLA to contribute to the project, since in the words of the project manager, Rémi:

> "CLAs make contributing to open source projects ridiculously bureaucratic and slow."

Some pull requests spark discussion, since a developer may want to include a feature that is deemed unnecessary by others, and in such a case, it may happen that the PR will be discussed in a meeting on the Godot developer IRC channel. The PR will only be merged after a consensus is reached.

Some developers have expertise in a certain area of the project and are allowed to commit directly if it falls within their field of responsibility.

Release Process

Development on the master branch is working towards a new stable release, and there is also work being done on the previous releases in a separate branch. After the big update from version 2 to 3, there were compatibility issues when users wanted to import projects into the new version of the editor and engine, therefore there are still people working on their games using an older version of the engine. Development on older versions generally is restricted to bug fixing, while in the master branch for the new releases new features and tools are constantly being added.

When a big release is approaching, a callout is made for developers to fix remaining issues. Sometimes before a stable release, a test build is released. These builds are announced on the blog, and anyone interested can test them and report issues.

The release schedule of intermittent updates is not determined and dependent on the severity of issues found and fixed.

Deployment Viewpoint

The Godot Engine offers support to build projects using the engine to the following platforms: Android, iOS, HTML5, macOS, Windows Universal, Windows Desktop and Linux/X11. To build the Godot Engine from the source code, the cross-platform build tool *SCons* is required, which in turn requires Python 2.x to run. If a user wants to build a project for a specific platform, they need a proper set-up for the target platform as well as an export template provided by Godot.

Some platforms have extra requirements to generate builds for them, as can be seen in this table:

Environment	Requirements for Exporting a Godot Engine Project to Platform
Android	Install ADK. Install JDK 6 or 8. Generate Debug Keystore File.
iOS \ macOS	Must export from a computer running macOS with Xcode installed.

For web exports (HTML5), running the export requires support for WebAssembly and WebGL 2.0 in the user's browser.

The export templates are used to set properties that are specific to a certain platform. For mobile platforms for example, you can set the screen orientation for the application to portrait or landscape mode. You can also manage asset compression and level of detail for the graphics. The export templates are required to create packages. In the Godot Editor you can easily install the export templates from the menu, and they can also be obtained from the download page on the website.

Supporting all the formats out-of-the-box is impractical, as it would bloat the core with a lot of logic people would not be using, and therefore platform-specific code can be found in the platform folder of the Godot Engine source code. For rendering graphics, the Godot Engine depends on the OpenGL ES 3.0 API, thus any devices that wish to execute projects made using the Godot Engine (this includes the Godot Editor) must have an OpenGL ES 3.0 compatible operating system and hardware.

Internationalization Perspective

The game market is a global one, therefore by including localization, games can reach a larger audience. One of the functionalities the Godot Engine offers is support for internationalization.

The internationalization system in Godot is based on the well-established gettext/PO files workflow, although using a tailor-made parsing and integration of the translated strings that does not rely on gettext. Strings that users will see can be wrapped in the TTR()-function. The supplied strings are used as keys for looking up alternative translations. When no translation is available, the function will return the original string. Translations can be imported into game projects as resources. The *Translation* class maps one string to another string, and the Translation Server singleton manages all the translations. Godot has a special importer for CSV-files and for .po-files.

The Godot Engine supports Unicode, so languages with diverse characters such as Chinese, Japanese, and Arabic can be encoded as well. The Godot Docs has a list of all the supported languages.

Translating the Godot Editor

The work of translating the Godot Editor is being crowdsourced to the community through Weblate, a free web-based translation tool that supports the use of GitHub for version control integration, with translations to 45 languages (including Pirate(!)) in progress as shown on the Godot project page on Weblate. Users can fill in, correct and confirm translations, and even add new languages. Weblate generates .po-files. Occasionally, one of the Godot core developers will synchronize the translations from Weblate with the git repository.

Technical Debt

When a coding team decides to achieve something using suboptimal means that only solve the imminent problem, what they are doing is essentially borrowing time and manpower from the future, since replacing it down the line will end up taking even more time and manpower than what the team borrowed. This concept is referred to as 'Technical Debt'.

Analysis of Technical Debt

Code Debt

Thanks to the excellent workflow model of the project, code debt is relatively rare in the Godot project. Most contributors assign themselves to an issue or feature, and then work to solve it, something which gives each pull request a clear goal. Moreover, the project manager, Rémi, with help from other senior contributors, inspect every pull request and conduct proper code reviews to ensure certain quality standards. Since the Godot Project is community-developed and not-for-profit, the developers do not need to rush production to meet deadlines, something which therefore results in less amount of 'hacks' that aim in only addressing something in the short-term.

As the lead developer and one of the original creators of the engine, Juan is very much concerned with keeping the core engine optimized and the code base maintainable, therefore he followed the SOLID principles during the development of the engine. The Single Responsibility Principle can be observed in individual classes, while other principles can be observed in the way the class inheritance and interface are designed.

High-Level Debt

The largest part of the code base is written in C++03 which by now is a relatively old version, however, to compile the Godot Engine, a C++11 compiler is required as some features of this version are used in the code.

Historical Analysis

As demonstrated in this issue, developers have discussed converting everything to fit more recent versions of the C++ language, however there are no plans to actually implement this anytime soon.

Just like the programming language issue, there has been some discussion about changing to another build system, however there are no plans to act upon it in the near future. One of the developers said:

> "It's hard to deny that a flawed but existing and working and reliable system is better than a potentially more efficient, yet mostly theoretical and difficult to (re)implement one"

Documentation Debt

Proper documentation is of paramount importance in every project, but more so in one as big and complex as a game engine. Documentation in this case is important for both developers and users, since it can assist them in understanding what the code is supposed to do, so adjusting or adding to it can be less complicated, and it can help game developers understand the use of the different functionalities available to them. So in turn, documentation debt can be detrimental to the manageability of a project, and especially of an open source one where everyone can make contributions.

Historical Analysis

Extensive documentation exists in the previously mentioned GitHub repository, however the main problem is that in many cases pull requests with no documentation are merged, something that will cause problems in the future.

Project Debt

As can be seen from previous sections, the backlog of issues continues to grow and there is no process in place to pay of this debt. The bulk of organizing this large amount of issues falls to the Project Manager, Rémi, who has other tasks and responsibilities related to Godot, and can be unavailable at times. While other developers have the privileges to add labels and milestones to issues, most of them are hesitant, believing Rémi and Juan to have a better judgment on these issues. This causes work to pile up, increasing the backlog. Having so few people carry so much responsibility is a technical debt, as it will be difficult to replace them when they are unavailable. Also, there is a need for more developers to contribute to the project, especially those with an expertise in some of the more specialized areas that require high level mathematics, such as rendering.

Historical Analysis

Efforts have been, and are being made to pay off this technical debt. When Rémi is overwhelmed by the workload, he has in the past used the mailing list to call out for help. On a semi-regular basis, the core developers get together on the development IRC channel to do PR reviews together, which is a good opportunity to have several developers give their opinion on changes that are under discussion.

To compensate for the lack of skilled developers, an approach that has been used in the past is to replace code written by Godot developers with better-performing third-party libraries, with an example being the physics system, which is being phased out of the project, now that the Bullet physics library has been fully integrated.

Part of the money raised by the Patreon campaign will be used to hire some of the skilled core developers of the project to spend more time working on the more difficult areas of the code base.

Solutions

Using an old version of C++ has already created some technical debt and could create a lot more in the future, especially since using an older version of a language always comes with the risk of code becoming obsolete. We recommend keeping the discussion going and perhaps begin to slowly port code over so that the amount of work necessary will be less, when the decision is finally made.

To avoid further documentation debt, we suggest that contributors are encouraged to document their pull requests, and in the case of new feature additions, we recommend not merging them until the authors provide sufficient documentation. For existing documentation debt, we suggest having a dedicated team of developers track undocumented features, and either contact the authors to provide documentation, or examine the code and write documentation directly.

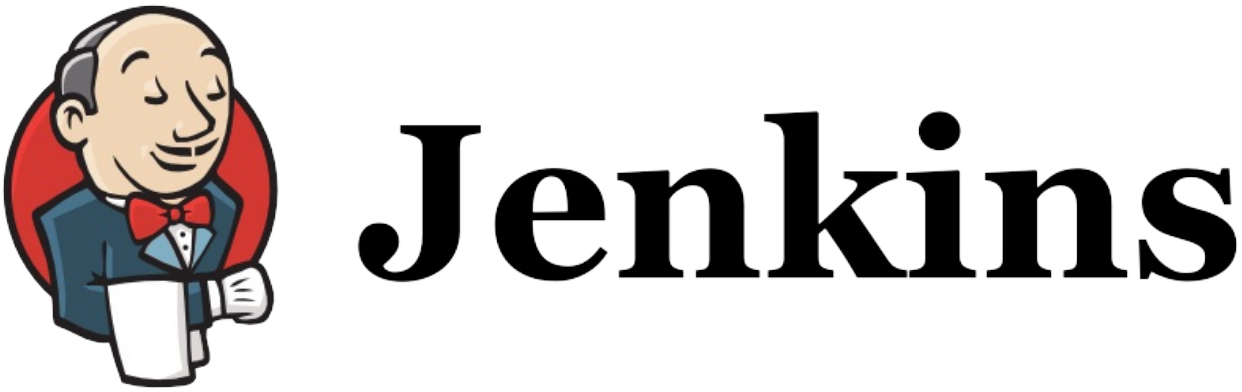
Certain older issues are not relevant anymore, so it would be prudent to create issue-testing teams, with the task of verifying whether they have already been addressed. If so, they can be closed, otherwise they can be assigned to the upcoming milestone so that developers will have a look at it when a new release approaches.

Conclusion

Game development is known for code being written in an unstructured manner as scope is changing constantly. A game engine on the other hand, is an on-going project that needs to have a well defined architecture, that is robust as well as flexible. It has to be able to provide tools so that game developers are free to explore different genres and features for their games. In our analysis we found the most important qualities for a game engine system to be stability, portability, maintainability and modifiability. Stability is maintained by constant testing by the user base. The code base is kept maintainable by diligent code review by the core developers, giving the project a high level of code quality despite being worked on mostly by volunteers. Long term solutions are prefered and quick fixes avoided as much as possible. The layered architecture for the Godot Engine separates platform-independent from platform-dependent code to facilitate portability to various platforms. Interfaces to the engine can be used so that developers can create modules that expand the functionalities of the engine making the engine very modifiable.

While many issues remain, we are confident that with the help of the excellent workflow, clear architectural model, and enthusiasm from the community, the Godot Engine will not crumble to its technical debts, but will instead, grow even better over time.

JENKINS: Build great things at any scale.



Contributors:

- Federico Fiorini (4743105)
- Jeroen Vrijenhoef (1307037)
- Ka Wing Man (4330714)

Abstract

Jenkins is the leading open source automation server, which provides an efficient and user-friendly way to support building, testing, deploying and automating any project. The aim of the entire project is to meet each developer's needs by providing not only the Jenkins automation server itself, but also a list of hundreds possible plugins which could be used to enhance Jenkins' capabilities. This chapter provides different views and perspectives of the project, with the aim to create the possibly most complete overview of the architecture itself and all the things that are involved around it. These views and perspective range from the stakeholders analysis to the developers perspective, highlighting the key architectural and functional features of Jenkins (and how it interacts with its plugins). Furthermore, we will conclude the chapter presenting an evolution perspective to highlight the key phases of the development of Jenkins, and a brief conclusion of what we think of the project itself.

Acknowledgments: We thank former team member Haris Adzemovic for his contributions to this chapter.

Introduction

Jenkins is a continuous integration tool, having over hundreds of plugins that provide support for automating tasks like building, testing, delivering and deploying the users' own projects, running tests to detect bugs and other issues and doing static code analysis, so that users can actually spend their time doing things machines cannot. [1]. The tool is a Java-based program with packages for Windows, Mac OS X and other Unix-like operating systems. The tool is accessed and configured on its web interface and the tool can be, as mentioned before, extended via its plugin architecture, being able to provide (possibly) infinite functionalities for Jenkins. Jenkins can also distribute work across several devices, making Jenkins tasks across multiple platforms faster.

The Jenkins project is open-source and though it has its main development team that does most programming, everybody can contribute and make pull requests to the project. Not only writing lines of codes, users can also help translating, make documentation and test the project.

In the Jenkins project, you might see some associations with Hudson, another continuous integration tool developed by Sun Microsystems (owned by Oracle Corporation). This is because Jenkins is a fork of Hudson, made in early 2011 [2].

This chapter provides an overview of the Jenkins projects, to show the reader how the project/community of Jenkins is built up. Next, the chapter identifies the stakeholders and integrators of the project, as well as a context view in which Jenkins is placed. Then, it provides both a functional and a development view of the project, explaining the structure of the project in terms of interfaces and

modules. The last part contains an overview of the technical debt of the project.

Stakeholders Analysis

Acquirers
Oversee the procurement of the system or product
The Governance board members acts as acquirers of the project. The board is involved in making the ultimate decision when the issues cannot be resolved by the community of the Jenkins project. The members of the board can be found on the Governance Board Page. The members (at the time of writing this document) are `R Tyler Croy`, `Kohsuke Kawaguchi` and `Dean Yu` [4] [5].

Assessors
Oversee the system's conformance to standards and legal regulations
The governance board also acts as a public representative of the project and therefore are most likely the ones that deal with standards and legal regulations [4].

Communicators
Explain the system to other stakeholders via its documentation and training materials
Jenkins also has a website with documentation, a blog, users mailing list, developers mailing list, a Wiki, Twitter account and so on. They could be used to communicate with stakeholders of this project [1].

There are also companies who offer support ranges from training for using Jenkins to consultancy on the software. One of these companies is Cloudbees, where the Jenkins creator `Kohsuke Kawaguchi` works.

Developers
Construct and deploy the system from specifications (or lead the teams that do this)
As this is an open source project, everybody may contribute to this project and thus the code contributors to this project can play the role of developers. These contributions include bug fixes and new features.

Maintainers
Manage the evolution of the system once it is operational
The maintainers are people who fix bugs and add new features to the project after it has been launched. They are also responsible for actual accepting and rejecting pull requests and pull request code review. Everybody who has been contributing with code belongs to the group of maintainers.

The top 5 people reviewers in 26-01-2018 – 26-02-2018. The ones in italics actually merged the pull requests.

1. *oleg-nenashev* 31
2. *daniel-beck* 20
3. *Jglick* 11
4. *Wadeck* 9
5. MarkEwaite 5

Support staff
Provide support to users for the product or system when it is running
Jenkins does not have an official support staff which you can contact if you need any help with Jenkins. Support can be given by the community of Jenkins instead.

Infrastructure admins
The Infrastructure admins have root access to several servers and build slaves that run on the domain `jenkins-ci.org` and the other sub-domains. Their tasks is to keep the aforementioned servers running, installing new software, coordinating mirrors, handling keys and certificates and make sure that the comunity keeps updating the Jenkins project with code. The admins also often appoint others to delegate some partial access to the system to complete some tasks. The list of admins can be found on the Infrastructure Administrators page [4]

Contributor
As specified before (and by Jenkins website), basically anyone can contribute to the project. This means not only write code for the core or its plugins, but basically anything related to the project itself, namely translators, documentation makers, testers, and so on.

That means that a *contributor* is not strictly identified as a developer or a tester, but could be anything at any time (there's no indication related to the fact that a contributor can switch its role inside the project). Therefore, in open source projects the contributor can be seen as a different kind of stakeholder with (often) the same functions as a developer, but maybe with less decisional power or who is part of the online community and not of the company/organisation.

Stakeholder involvement

Another way to classify the stakeholders is to classify them by the power to be able to change the system and by their interest in the system.

1. Official Jenkins members
   The people who are officially associated with Jenkins in any way.
2. Jenkins Community
   The Jenkins community is the community that helps contribute in Jenkins in any way (e.g. code, documentation etc).
3. Developers community
   Developers are the users that uses Jenkins for their own projects.
4. Communicators
   See `R&W Stakeholders` section

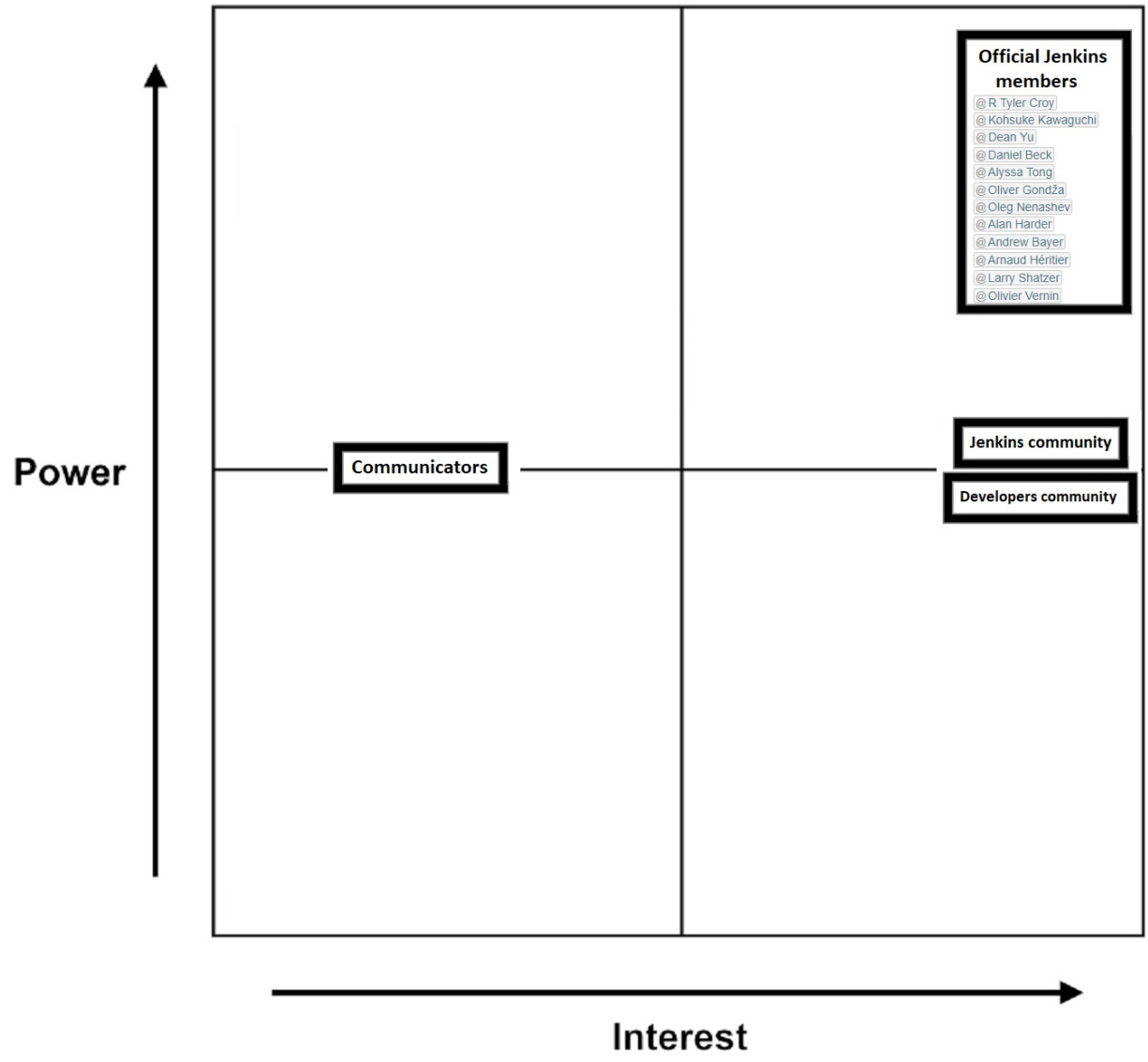


Fig. 1 - Power-Interest Grid for Jenkins' stakeholders

Integrators

It's possible to identify **Daniel Beck** and **Oleg Nenashev** as the main integrators [6], as they're involved in every pull request; moreover, their work is coadiuvated by a restricted (and private) group of contributors, namely the *jenkinsci/code-reviewers*, which helps the two major integrators in the reviewing process. The integrators decide whether to accept a change based mainly on code quality and both project and technical fit; in case the code has some problems, they usually propose changes using long discussions and reviews, in which usually ask the contributor to explain why he/she has made that choice. Last but not least, they focus on testing: in fact, in order to accept a pull request, that has to pass from an automatic code checking and testing (named *continuous-integration/jenkins/pr-head*) which highlights if the code has failures and, in that case, it needs a further review. Most of the times, if the code proposed doesn't pass the tests, no merge is performed. The acceptance time ranges generally from one day to one week, depending on how fast the debate for a pull request goes and on how "important" the contributor is: if it's a top developer, the discussion would take less time and usually those PRs are closed within one day.

Context View

System Scope

- Jenkins is a self-contained, open source automation server built in Java which can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software. It works along Docker, a container-virtualisation software, and JRE (Java Runtime Environment). Both should be installed while trying to install and setup Jenkins.
- It works with all operating systems: Windows, MacOS and other Unix-based OSs.
- It can be enhanced with many plugins, extending its functionality for every user's demand; they're provided and can be downloaded through the Update Centre. [7] They can also be browsed using the plugin browser. [8]
- It can either run on a single PC or a distributed system, in order to spread the computational load.
- Through special files (*jenkinsfile*) or the Blue Cloud interface (included in the basic plugin package) it is possible for users to define and manage their own workflows ("pipelines").
- It's also possible to access details such as instance log files or console outputs to see if the build/execution/test/etc. is working as predicted
- Users can be notified during the execution of a pipeline.

External entities involved

Jenkins (more precisely, its core) have to interact with many entities, which are both internal and external systems. The most important internal systems with which the core interacts are the Jenkins plugins, as many use cases for the system involve at least some basic plugins. There's a frequent interaction between core and plugin while the system is working, and it's all managed by the code and therefore completely automated (so it's basically transparent to users). These plugins act as both service and data providers and consumers, as they need the core for their functionality and they extent the core's capabilities or features. Another internal system is the so-called *Jenkins Remoting* [9], which is a library and an executable Java archive which implements the communication layer in Jenkins. This system is essential for the correct operations of the core as it provides all procedures and protocols that allow communication between all system's components through a network. The core and the remoting system interact frequently and it's all transparent to users as well.

There are also several external systems with which the core interacts:

- [*Jenkins Infrastructure*]
  Since it's an independent and open source project, the entire infrastructure on which Jenkins is running is maintained along with the project. It can be considered as external because it's not directly part of the core itself, but it interacts with it as the core needs the

infrastructure to work. Moreover, it's needed in all use cases that require Jenkins to be distributed. [10] It's possible to have failures in the infrastructure, but there might be redundancy measures to preserve Jenkins' functionalities.

- *Github and other communication means*
  GitHub is the principal tool for project management, versioning and communication used by Jenkins. It's also the primary source for the code, and all contributors must use it to make their own modifications (either fixes or new features). Jenkins communicates with Github in several ways, for example if the results of the builds of other projects that uses Jenkins are shown on the Github webpage. Github is also used for integrating builds. Other communication means include mailing lists, forums and also a dedicated Issue Tracker website. [11]
- *Java and Docker*
  Java and Docker are required for the installation of Jenkins, and therefore they can be mentioned as two major external systems with which Jenkins interacts. Java provides the framework and the programming language, while Docker provides a virtualisation means [12]. They're both vital in order for the system to work, and their interactions are completely hided from the users, as they're all automated by the code.

Context View Explanation

The final context view can be seen in the diagram below:

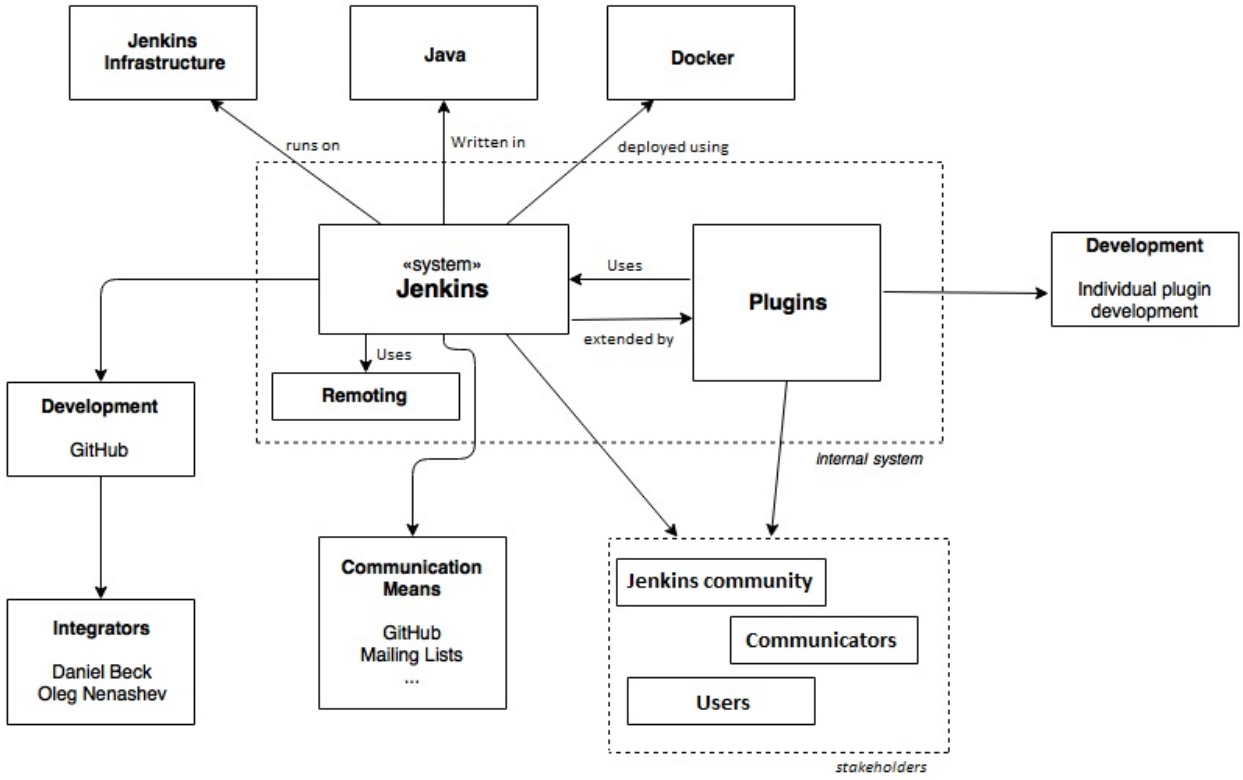


Fig. 2 - Context Diagram for Jenkins

As it has said before, Jenkins and its many plugins live symbiotically. There are very few use cases where anyone would use Jenkins without any plugins at all. The only feasible use cases we have come up with are related to development of Jenkins itself. All other use cases include at least some basic plugins such as the Green Balls plugin, which colors the icons of tests' status in more intuitive colors than the core package offers. Since most users rely on plugins, the Jenkins core team is very careful not to break plugin compatibility while upgrading the core since this could lead to user abandonment if their workflow is interrupted. The plugins themselves of course also rely on Jenkins as without a working core, they're useless.

Communication is spread out over many entities as the bigger plugins often have their own websites, forums, mailing lists etc. Jenkins itself mostly uses GitHub, mailing lists and its own website [1]. Documentation is handled by the open source community and available through the same channels as the communication.

Development of Jenkins is done by the open source community. Oleg Nenashev and Daniel Beck are two identified main integrators who approve and merge pull requests. Development of plugins is individual for each plugin.

Functional View

This chapter focuses on the architectural elements of Jenkins that deliver its functions at runtime

Capabilities

The key capabilities offered by Jenkins are:

- Continuous integration and delivery
- Extensibility
- Ease of configuration
- Maintaining backwards compatibility
- Dedicated CLI for different types of users

In that sense, Jenkins is expected to provide an efficient service that helps developers in building, testing and delivering their projects. In order to provide a better user experience, and to help users to personalize the building processes, Jenkins is designed to be extensible, in the sense that it's possible to install multiple plugins and configure them easily to meet the developers' needs. Next we point out some of the architectural principles that might have been used in designing Jenkins:

- Ease of extension
- High separation of functionalities and concerns
- Loose coupling for core functionalities

Interfaces and structure

The following image gives a high-level view of the runtime model of Jenkins:

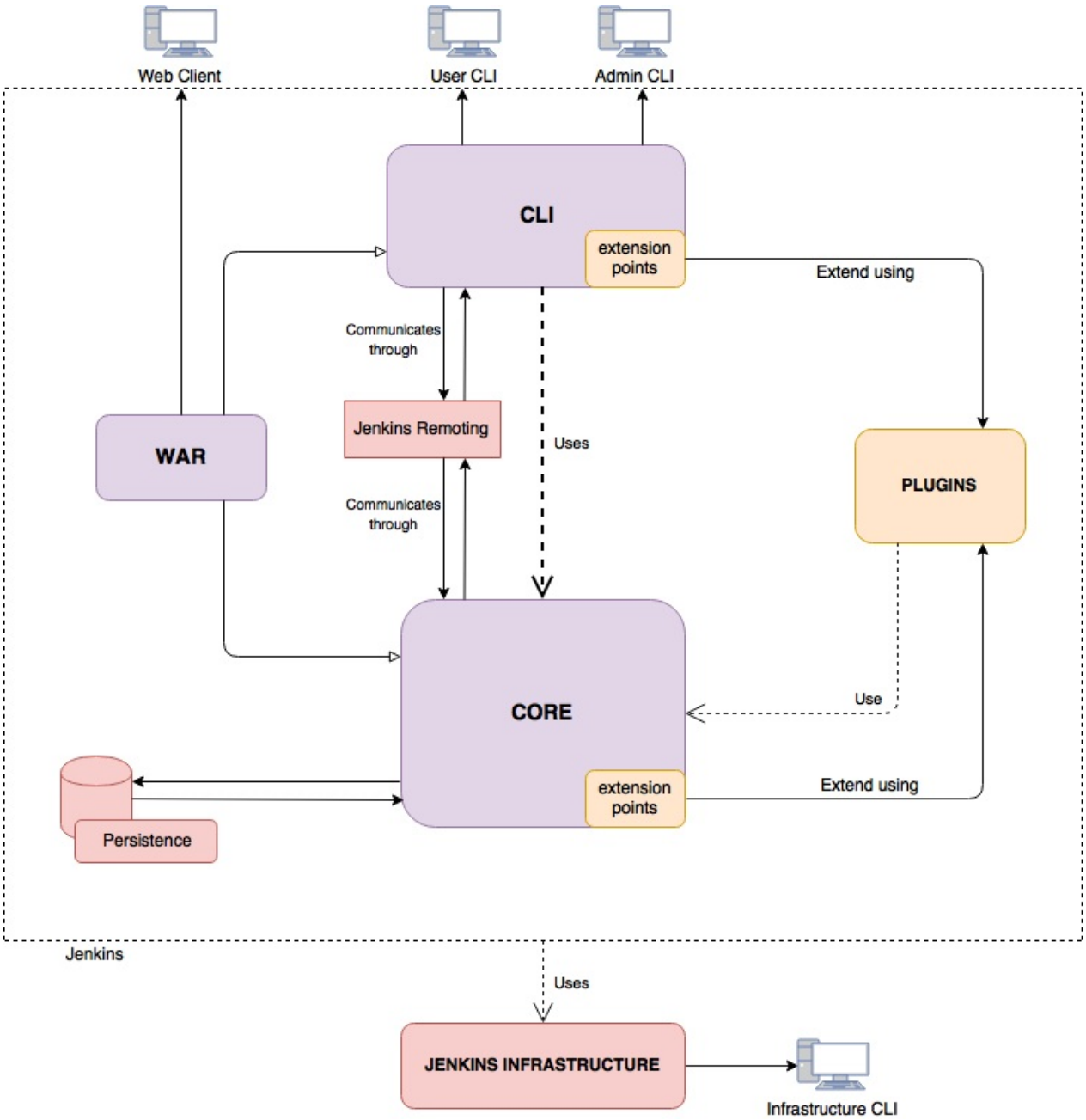


Fig. 3 - Functional interfaces and entities for Jenkins

It's possible to identify some external interfaces, such as:

- extension points
- web client, user/admin/infrastructure CLI:
- remoting

In particular, extension points are used to extend core's and cli's capabilites and create new plugins, which will need the core itself in order to work and to provide new functionalities and tools. They're basically interfaces and abstract classes or methods which can be extended for whatever use. *Jenkins Remoting* is used to make the CLI and the core communicating, especially in a distributed environment, and it's actually not part of the Jenkins project itself: it only provides communication APIs and procedures. Last but not least, there exist many user interfaces, each one of them designed to be used for specific purposes: web client and user CLI are used by normal users to install, upgrade, manage and work on Jenkins, while the admin CLI is used by system administratos to monitor Jenkins and its plugins. The infrastructure CLI is normally used by the maintainers of the *Jenkins Infrastructure sub-project*, and it's function is to ease the monitoring of all the components that help Jenkins to work.

Development View

This chapter is focusing on the development view of the Jenkins project.

Module Structure

Jenkins is a large project, with a complex architecture consisting of several thousands of lines of source code spread over hundreds of files: in order to have a better understanding of the architecture, as well as for ease of testing and maintenance, the source code is organized into four different modules, each one of them providing a specific functionality.
The four modules are:

- **cli** it provides the command line interface for Jenkins
- **core** the core source code for the project
- **test** unit for functional tests on the core code
- **war** responsible to create the .war file

The first three modules can also be divided further into submodules. In particular, both **core** and **cli** have their own test units inside the modules, which provide specific test functions (in contrast with the more general testing units provided in the **test** module).
Furthermore, **core** module is divided into two logical modules, named *Jenkins* and *Hudson*, which are further divided into modules, each of them providing a different functionality to the core itself. In the following picture there's a quick overview of the main modules in deep:

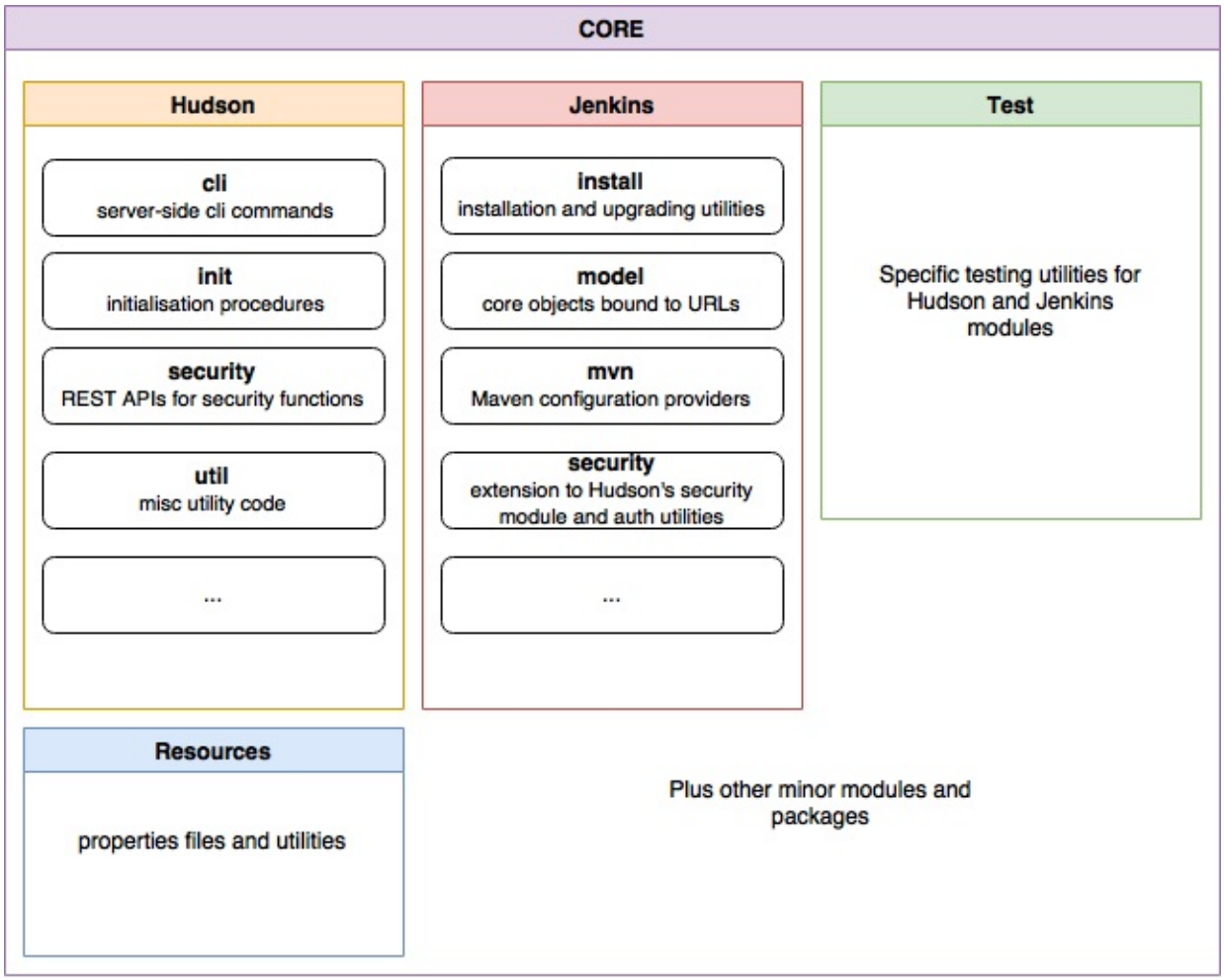


Fig. 4 - Module Structure

Module dependencies

The following picture represents a high-level and conceptual view of modules dependencies, analyzed through code inspection.

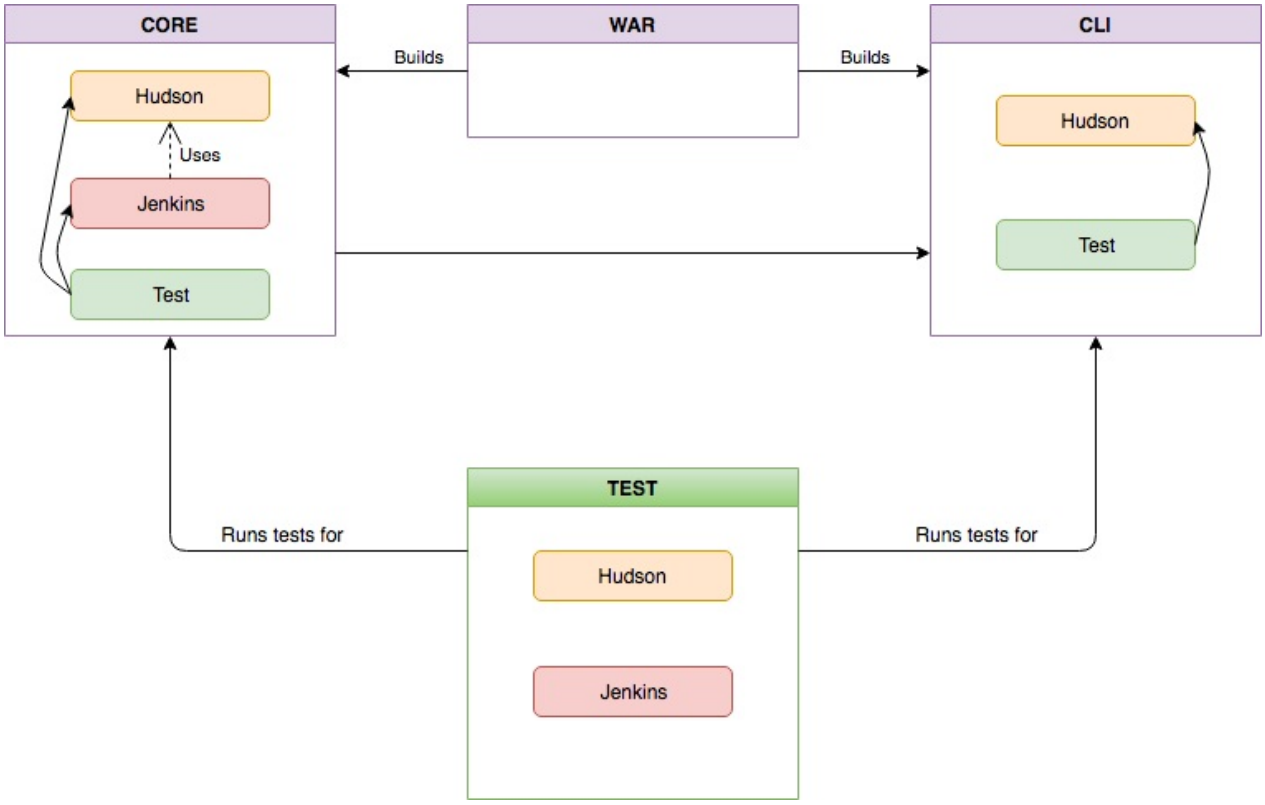


Fig. 5 - Jenkins module dependencies

All dependencies can be found also looking at the `pom.xml` files contained in each module, which are Maven files concerning the project's properties, as well as single modules properties. In particular, in each of them there's a detailed list of modules (internal to the project or external libraries) on which a certain module depends. It's important to point out that there's an explicit relation between the *Hudson* and the *Jenkins* submodules: in particular, lots of function used in the latter invoke functions contained in the former one.

An example of how these dependencies work can be seen in the following picture, which refers to code analysis done using IntelliJ IDEA and emphasizes when these dependencies are needed.

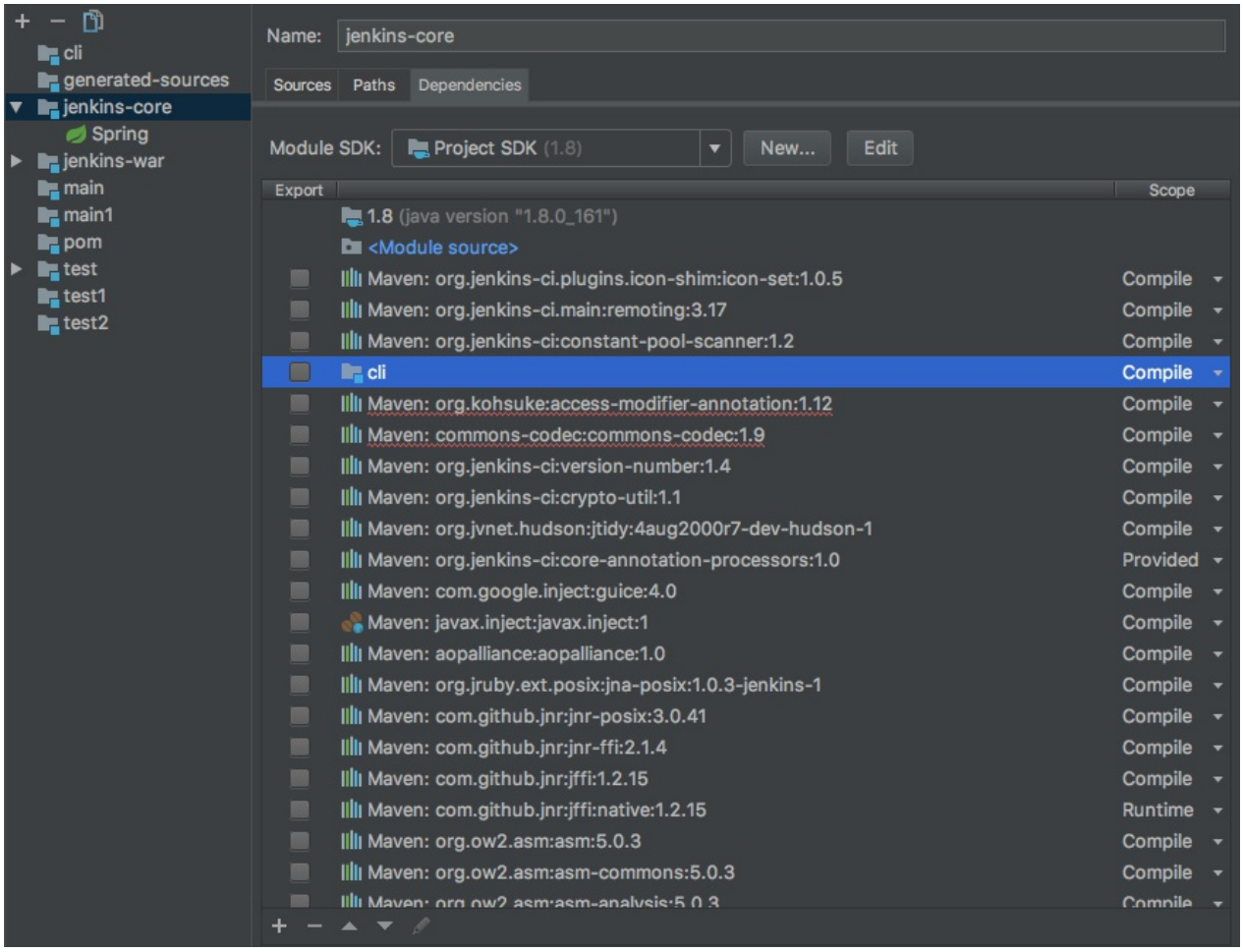


Fig. 6 - Example of dependencies depicted by IntelliJ IDEA

Common Design

This section focuses on common processing inside the Jenkins project. It's further divided into subsection for each important area

Jenkins as common processing

Jenkins plugins can be considered the most important part of the entire Jenkins ecosystem, as they provide augmented core capabilities and new tools or utilities to simplify user interaction or processes. In that sense, it's common to say that plugins are made to meet specific projects' needs. While their development is not provided by Jenkins core developers, it's possible to state that core and plugins are somewhat entangled: in that sense, Jenkins core works as the starting point for developing and building new plugins, and they all use its core functions to work; moreover, core developers themselves have made the core's source code in a way that it has some extension points, from which it's possible to create new plugins and functions.

External libraries and dependencies

Jenkins modules heavily depends on third-party libraries in order to work, so it's important to define the most important dependencies. Mostly all modules depend on a specific set of libraries, including:

- **JUnit4** [13] , **Mockito** [14], **Powermock** [15] for testing purposes
- **Apache Commons** [16] includes all reusable code for I/O purposes, authentication methods, XML scripting, configuration files, logging purposes, ecc.
- **Ant** [17] and **Maven** [18] for project build and management

- **Jenkins Remoting** [19] for communication layer libraries and functions
- **GitHub APIs** [20] to manage GitHub compatibility with Java (directly developed by Kohsuke Kawaguchi)

Required design constraints

The philosophy of the project is to have low barriers of entry, so basically anyone can contribute to the project. However, new contributors have to adhere to the standards explained in the `contributing.md` file, which basically provides a template for the pull requests and an overview of the procedures. [21] Furthermore, the developing team tries to adhere to the Sun Coding Convention [16], but they don't require additional code formatting rules; they also require contributors to look for an overall ease of code understanding, testing and extensibility while making a contribution, as well as looking to retro-compatibility (so it's not safe to remove completely some functionalities from the codeline). It must be pointed out, however, that they don't force contributors to use these general coding conventions. Citing the *governance document*:

> With that said, we do not believe in rigorously enforcing coding convention, and we don’t want to turn down contributions because their code format doesn’t match what we use. So consider this informational. [4]

Concerning plugins, as the developing processes are separated, there are no constraints at all, but they can be required by each plugin's developing team.

Internationalization

Jenkins is used almost everywhere in the world, so having a correct translation for messages, client interfaces and software itself enhances the user experience and also helps contributors from different countries to work on the software. The internationalization extension is therefore a crucial aspect of the overall development of Jenkins, and translators are warmly encouraged to contribute. However, there are a few common aspects that translator need to know:

- *Translation Tool* and *Translation Assistance Plugin* are possible tools to make internationalization contributions [22]
- There are three types of resources that need internationalization:
    1. *messages*, which have to be put into the related `Message.properties`
    2. *Jelly messages*
    3. *static HTML references* for the website

Once the translator has done its work, it can submit it like normal code contributions.

Codeline

Source Code Structure

The source code is organized according to the module structure explained before, and in each folder we can see the `pom.xml` file needed to analyze dependencies and properties for that particular component.

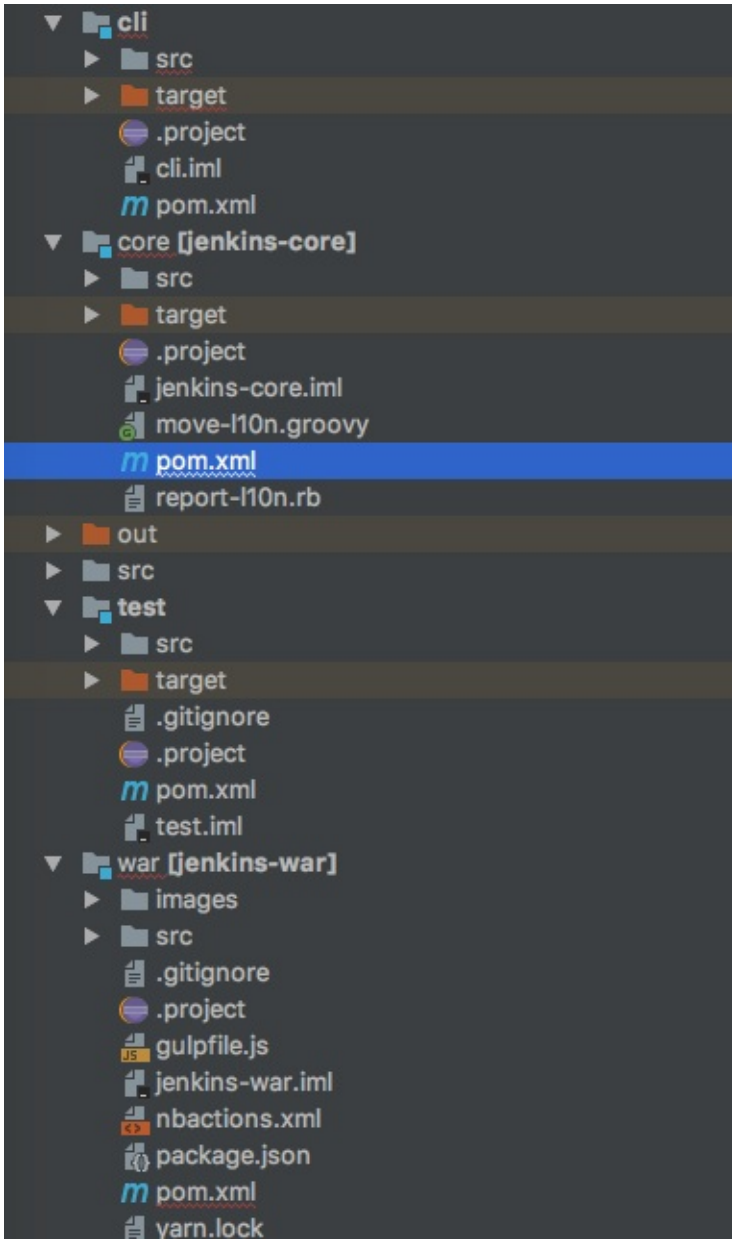


Fig. 7 - IntelliJ module representation for Jenkins

This simple approach allows an easier understanding of where to put the source files, and the naming conventions used for packages and directories further simplify the maintenance process, as it's possible to locate every file with a small effort.

Build, Integration and Test approach

Jenkins uses a peculiar approach when it comes to build and test new features or fixes, because it uses Jenkins itself to automate the process. In that way, developers can have direct feedback on the software's capabilities and functions, and it might be helpful to spot new bugs or possible performance improvements. In addition, the building process is further automated using Maven, and the testing units are made using automated tools such as the *Acceptance Test Harness*. [23]

Release Process

In the Jenkins project, we can define two types of releases that can be made:

- **Weekly Releases**
  Used to deliver bug fixes and new features rapidly to users and plugin developers who need them. Usually these releases involve small or minor changes, and usually contains only a few of them.

- **Long-Time Supports (LTS) Releases**
  These releases are made for more conservative users, which don't need frequent updates in terms of features or minor bug fixes, and they use the same concepts applied for Ubuntu's LTSs. Usually this kind of release contains major or crucial bug fixes, and they are made to ensure a stable environment in which use Jenkins. The process, described briefly in the LTS download page, involves special types of pull requests, which are called *candidates* and are tested for several weeks before being published. [24]

Technical Debt

For analyzing the technical debt, we have used static analysis tools `CodeFactor` and `Findbugs` [25][26]. Along with static analysis we have also looked at the test-suite and Todo's. Using `IntelliJ IDEA`, we have imported the Jenkins project before to do a contribution for D2. As IntelliJ IDEA can also be used for analyzing this, this IDE was also used for analyzing coding style.

Code Quality

CodeFactor

CodeFactor analyses every source file contained in the repository independant on what type of language is used. Overall a grade of B- (Good) is given to the master branch of Jenkins. The most issues were found in Style and Maintainability categories. Where most of the problems involve unresolved warning statements and unused variables.

Findbugs

The results for the Findbugs analysis for the **core** and **cli** modules are condensed in the table below. The most prevalent warnings in the **core** module involve either bad practises, dodgy code and even security related warnings. Examples of bad practises that are common are shadowing of superclass names by base classes, incorrect handling of double/float values and improper exception handling (ignoring exceptions). Examples of dodgy code are potential integer overflows and null pointer dereferences (which arguably should also be security warnings). The security warning is related to HTTP header parameters in server responses that contain unsanitized user input. Making it possible for a user to insert arbitrary header fields in the response by including carriage returns/line breaks in certain user input.

The high priority warnings for the **cli** module are all internationalization warnings, related to reliance of the code on default encoding. This could lead to problems when the project is compiled on systems that have different locale settings, as encoding is different. The dodgy code warning involves a redundant nullcheck.

Metric	core	cli
High Priority Warnings	96	9
Medium Priority Warnings	421	1
Total Warnings	517	10

Historical debt

For the historical debt we decided to analyze past releases of Jenkins over a time period of ~6 months from August 2017 to Februari 2018. Unfortunately CodeFactor does not support analyzing releases (only branches), so we opted to use the Findbugs plugin from `IDEA IntelliJ` to get an idea of how the technical debt related to buges changed over time.

The amount of bugs found found by Findbugs per category of bugs over time in the **cli** module are shown in the table below. As can be seen there is only 1 internationalization warning removed in a 6 month period. The dodgy code (redundant null check) is never fixed.

Warning Type	8-2017	9-2017	10-2017	11-2017	12-2017	1-2018	2-2018
Internationalization Warnings	10	10	9	9	9	9	9
Dodgy code Warnings	1	1	1	1	1	1	1
Total	11	11	10	10	10	10	10

The results for the **core** module is shown in the table below. Overall the amount of bugs found is actually increasing over time. Some subcategories decrease, such as Bad practise warnings. But when they do other bug categories such as correctness and dodgy code warnings actually increase in count. This indicates that while the project is actively being developped they also amount more technical debt.

Warning Type	8-2017	9-2017	10-2017	11-2017	12-2017	1-2018	2-2018
Bad practice Warnings	137	137	137	136	130	137	129
Correctness Warnings	137	138	138	142	150	140	150
Experimental Warnings	2	2	2	2	2	2	2
Internationalization Warnings	52	52	52	52	52	52	52
Malicious code vulnerability Warnings	90	90	90	88	89	90	89
Multithreaded correctness Warnings	21	21	21	21	21	21	21
Performance Warnings	19	19	19	19	19	19	19
Security Warnings	1	1	1	1	1	1	1
Dodgy code Warnings	435	436	437	448	457	443	457
Total	894	896	897	909	921	905	919

Test Coverage

This section contains the analysis of the existing testing debt for the Jenkins project. Every module in the Jenkins project structure has its own directory with a test suite. However, the bulk of the functional testing for the **core** and **cli** module is done in the **test** project. The test suites that are located in the **core** and **cli** modules are still being used and contain mostly unit-tests for their respective parent modules. These former testsuite contains 6 unit tests and the latter contains 3567 unit tests. Combined with the functional test suite from the **test** module which contains 9311 tests, makes for a total of 12884 tests. We did not take the javascript test suite from the **war** module into consideration. The **test** module also contains integration tests that test the interaction between classes.

The overall test coverage results for the **core** and **cli** module are summarized in the table below. For brevity, the individual coverage results for modules are left out.

Package	Class, %	Method, %	Line, %
all classes	77.8% (1849/ 2376)	59.8% (8468/ 14167)	56.3% (28147/ 49986)

When looking at the overall coverage results the line coverage percentage of 56.3% looks fairly decent. However when looking at the coverage at the module level it becomes apparent that some classes are very well tested, while others are not tested at all. For example the **core** module is completely lacking in unit tests for system management functionality such as file streams and there is a lack of testing of security related code such as the LDAP authenticator. It is also important to note that most tests involve unit tests that test a single method/class as opposed to integration tests.

Possible Improvements

The results from all previous analysis indicate that there is a lot of technical debt in the Jenkins project, and that the debt touches on multiple aspects from the project. Looking at the bugreports from `IDEA Intellij` , `CodeFactor` and `Findbugs` the biggest issues are related to either dodgy code practises (that lead to security vulnerabilities in the code) and the reliance on default encoding. Code duplication and code redundancy is also a big issue, indicating that a lot of code should be refactored.

The Jenkins **war** module, which contains the website frontend also has a lot of potential improvements. Biggest problems identified here by `CodeFactor` are missing gradient files for CSS, leading to bad browser support, lots of overqualified CSS elements and unused variables. The HTML code also contains a lot of deprecated tags and there is a big reliance as well on default encoding. The web frontend is also a big source of technical debt and should be updated or either completely rewritten.

Looking at the test coverage for both the **core** and **cli** project, it is obvious some parts of the code are well tested while other parts are not tested at all. Also adding more integration tests is also a big area of improvement.

Evolution perspective

Originally, Jenkins was developed as the Hudson project, a open-source project developed in Java by Sun Microsystems [27]. The development began in the summer of 2004 with the head developer being Kohsuke Kawaguchi, who worked for Sun Microsystems and its first release was in February 2005.

After Oracle acquired Sun Microsystems, and therefore also Hudson and its trademark ownership, the company insisted on certain changes to the code, infrastructure decisions, process etc. which Kohsuke and other prominent members of the Hudson community disagreed to do so. Because Oracle had the trademark ownership of the name Hudson, it could at any time revoke the project's right to call itself Hudson.

After the issue, votes were made to change project name from Hudson to Jenkins. Oracle, however, decided to continue the Hudson project under the name of Hudson. Thus, Jenkins and Hudson continued as two independent projects.

Looking from the split onward, Jenkins has been continuously update by the Jenkins community, with 759 releases being released since the initial release of Jenkins in February 2011 version 1.0.0 to the latest release in April 2018 version 2.114 at time of writing this chapter. Major changes have been made in version 2.0, released in April 2016. [28][29][30].

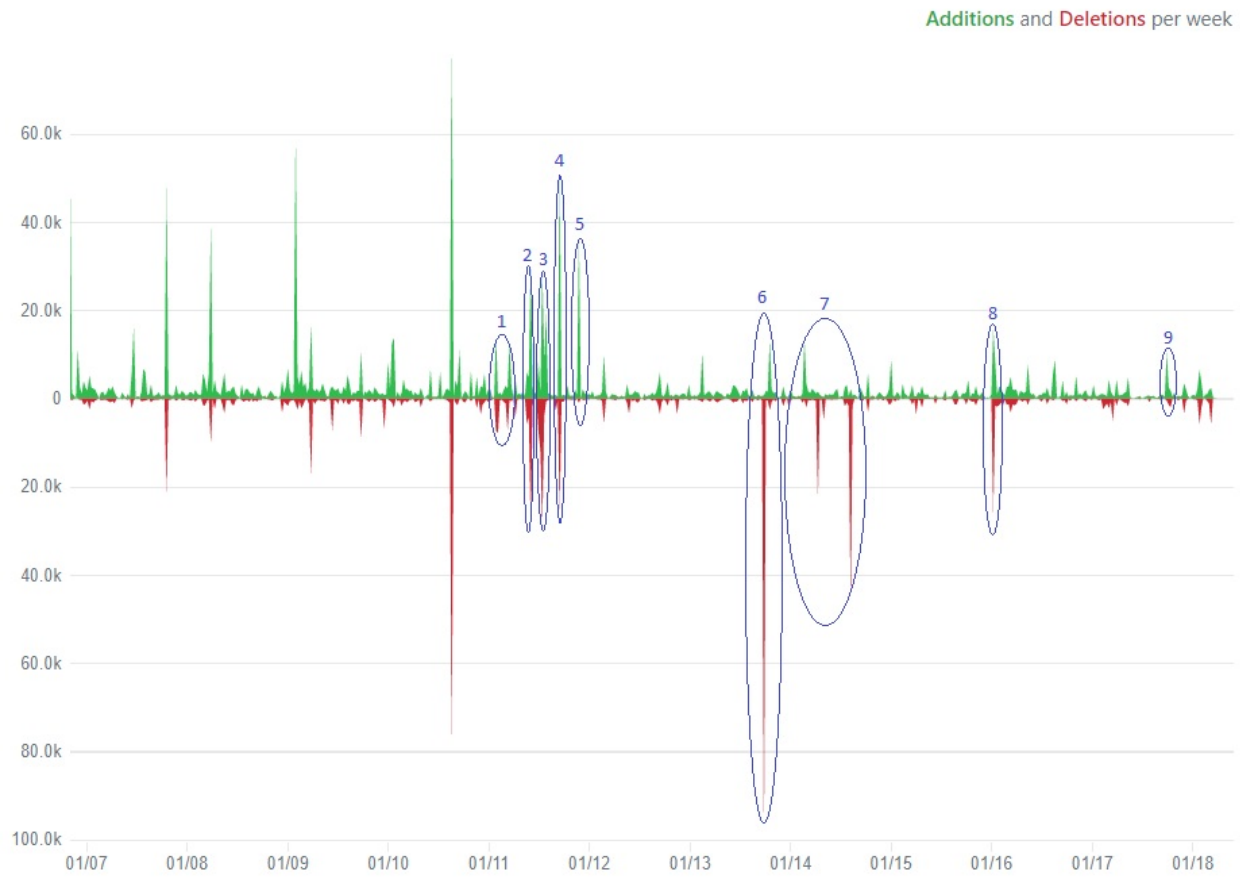


Fig. 8 - Code Frequency diagram

In the graph, there were already commits even before the Jenkins and Hudson split issue. These commit might be old commits from Hudson before the split, but made to the Jenkins repository after it was created. The time of commits can be changed easily, not reflecting the real time of committing.

Other major releases can be seen in the peeks:

1 - 6. Version 1.395 to 1.402, 1.410 to 1.420, 1.421 to 14.427, 1.428 to 1.430, 1.438 to 1.441, 1.526 to 1.537: Lots of bug fixes and small new features. Some of these features are Hudson changes to Jenkins and third-parts libraries upgrades.

1. Added infrastructure for moving items into/out of folders. Addition of a new API. Updates of libraries. UI redesigns.
2. Version 2.0: This version requires Servlet 3.1 and removed AJP support when using embedded Winstone-Jetty container, other implementations. Improvement of documents. Lots of small bug fixes and small new features.
3. Version 1.530 to 1.541: Core of Jenkins migrated from Java 5 to Java 6. Lots of small bug fixes and small new features.

There are still a lot of issues, 1108 critical issues where no solution has yet been committed. With more issues being created than fixed, it seems more contributors are needed to shrink the amount unfixed issues.

Conclusions

After all the analysis we did on Jenkins, we can conclude that unless its complexity it's a very well-organized project when it comes to the source code structure, making it easy to build, test and deploy. Furthermore, having a community of contributors, testers and maintainers which gets bigger and bigger every year contribute to the overall improvement of the project itself and its plugins at each new release; this also makes Jenkins one of the biggest open source projects you can possibly find on GitHub, as shown by the impressive number of repositories and forks of the main repo. The analysis we performed during the last weeks also helped us to understand that, despite a good structure of the code, there are lots of possible improvements to the project when it comes to bug fixing and test coverage, as pointed out in the potential improvements section. Nevertheless, Jenkins remains a leading automation server, and

its being open source surely helped the great spreading this software has, although it also stopped further improvements concerning test coverage and overall code quality (this could be seen as a direct consequence of being open source, as well as having low entry barriers for new contributors, but in general it's not much of an issue yet).

References

[1] Jenkins Homepage. https://jenkins.io/
[2] Hudson's future, blog post by Andrew Bayer. https://jenkins.io/blog/2011/01/11/hudsons-future/
[3] Jenkins governance board. https://wiki.jenkins.io/display/JENKINS/Governance+Board
[4] Jenkins governance document. https://jenkins.io/project/governance/
[5] Jenkins commercial support. https://wiki.jenkins.io/display/JENKINS/Commercial+Support
[6] Integrators, blog post by Georgios Gousios. http://www.gousios.gr/blog/How-do-project-owners-use-pull-requests-on-Github.html
[7] Jenkins update centre. https://updates.jenkins.io
[8] Jenkins plugins browser. https://plugins.jenkins.io
[9] Jenkins Remoting project homepage. https://jenkins.io/projects/remoting/
[10] Jenkins Infrastructure project homepage. https://jenkins.io/projects/infrastructure/
[11] Jenkins JIRA Issue Tracker. https://issues.jenkins-ci.org/secure/Dashboard
[12] Docker and Jenkins Homepage. https://jenkins.io/solutions/docker/ [13] JUnit4 Homepage. https://junit.org/junit4/
[14] Mockito Homepage. http://site.mockito.org
[15] PowerMock GitHub page. https://github.com/powermock/powermock
[16] Apache.org Homepage. https://commons.apache.org
[17] Ant Homepage. http://ant.apache.org
[18] Maven Homepage. https://maven.apache.org
[19] Jenkins Remoting project homepage. https://jenkins.io/projects/remoting/
[20] GitHub APIs, Kohsuke Kawaguchi. http://github-api.kohsuke.org
[21] Contributing.md file on jenkinsci/jenkins. https://github.com/jenkinsci/jenkins/blob/master/CONTRIBUTING.md
[22] Sun Coding Convention. http://www.oracle.com/technetwork/java/codeconvtoc-136057.html
[23] Jenkins acceptance test harness. https://github.com/jenkinsci/acceptance-test-harness
[24] Jenkins wiki page. https://wiki.jenkins.io/display/JENKINS/
[25] Jenkins LTS download page. https://jenkins.io/download/lts/
[26] CodeFactor Homepage. https://www.codefactor.io
[27] FindBugs Homepage. http://findbugs.sourceforge.net
[28] Jenkins release 1.936 blog post. http://jenkins-ci.361315.n4.nabble.com/Jenkins-1-396-released-td3257106.html
[29] Jenkins changelog (old version). https://jenkins.io/changelog-old/
[30] Jenkins changelog. https://jenkins.io/changelog/

Kubernetes - Container Orchestration

By Enreina Annisa Rizkiasri, Francisco Morales, Haris Suwignyo, and Mohammad Riftadi. *Delft University of Technology*

Table of Contents

Introduction

Kubernetes is an open source system for managing containerized applications across multiple hosts; providing basic mechanisms for deployment, maintenance, and scaling of applications [1]. After Google and the Linux Foundation established a partnership and announced Kubernetes 1.0 in 2015, Kubernetes was taken as the flagship project under the umbrella of the Cloud Native Computing Foundation (CNCF) [2]. Nowadays, it is one of the most used orchestration system for containerized applications, with a large base of users, partners, and an active development community.

Kubernetes provides a centralized management environment for containers, microservices, and cloud platforms. It orchestrates computing, networking, and storage infrastructure on behalf of the user. Thus, it combines the simplicity of Platform as a Service (PaaS) with the flexibility of Infrastructure as a Service (IaaS), enabling portability across infrastructure providers.

In this chapter, we cover who is involved in the development and growth of Kubernetes; we describe its context view and its functional building blocks. Afterwards, we dive into the development practices inside Kubernetes, describing its module organization and common designs. We also analyze the presence of technical debt in Kubernetes project and how it has evolved. Finally, we present our conclusions regarding the architecture of this project.

Stakeholders

Identifying Stakeholders

In order to understand the stakeholders' interests around such a large project as Kubernetes, it is necessary to introduce the Cloud Native Computing Foundation (CNCF) first. CNCF was born from a partnership between Google and the Linux Foundation when Kubernetes 1.0 was announced in 2015 and considered as its flagship [3]. Since then, many cloud computing and IT industry players plus other related organizations have joined CNCF to incubate, develop, and maintain an ecosystem of cloud projects under a shared, common vision. CNCF members are distributed in Platinum, Gold, Silver, End-User, Academic and Non-profit membership levels. Leading CNCF members, which include Google, Docker, IBM, VMware, Cisco, Red Hat, and Oracle, own a Platinum membership among others.

To identify Kubernetes stakeholders, we follow the categories from the book Software Systems Architecture by Nick Rozanski and Eoin Woods [4]. The main Kubernetes stakeholders are found as part of the CNCF or the Kubernetes general community. CNCF's charter defines several organization divisions with representatives from different CNCF members. Stakeholders' roles can be matched to these organization divisions and to other groups derived from the CNCF members and the general Kubernetes community as described next.

There are three high-level divisions inside CNCF, which is a key aspect of stakeholder identification: Governing Board, Technical Oversight Committee (TOC), and the End User Technical Advisory Board (TAB). The Governing Board (**Acquirer, Assessor**, mostly formed by Platinum CNCF members) is responsible for budget decisions, marketing (**Communicator**, through a Marketing Committee), and defining and enforcing policies regarding the use of trademarks and copyrights of the foundation. The Technical Oversight Committee (**Assessor**) is expected to define and maintain the technical vision for the CNCF, align, remove or archive projects, define common practices to be implemented across CNCF projects, among other duties. The End User TAB (**User**) receives the input from the End User community (CNCF members and no members) to coordinate and drive activities important to the users and consumers of CNCF projects. Some specific Kubernetes end users include Huawei, Bla Bla Car, ebay, Goldman Sachs, and Philips [5].

Besides CNCF organizations, Kubernetes attracts thousands of contributors worldwide which coordinate their efforts through online platforms like Github, Slack, and StackOverflow (**Suppliers**). Kubernetes' development is in charge of Special Interest Groups (SIG). SIGs range from Architecture (**Maintainer**, ensures architectural consistency over time), Product Management (**User**, manages user requests and feedback), testing (**Tester**), to specific implementations for service providers like AWS, GCP, Azure (**Production Engineers**, **Support Staff**, and **System Administrators** often belong to service providers).

Based on the number of commits, we have identified the top 5 developers on Github:

- smarterclayton,
- brendandburns,
- wojtek-t,
- deads2k, and
- caesarxuchao.

Other Stakeholders

Table 1 identifies four special types of Kubernetes stakeholders [6].

Table 1. Kubernetes special stakeholders

Stakeholder Type	Description
Certified Service Providers	Service providers with deep experience helping enterprises successfully adopt Kubernetes. E.g. Accenture
Certified Platforms & Distributions	Software conformance ensures that every vendor’s version of Kubernetes supports the required APIs. E.g. Cisco
Technology Partners	Integrations and plugins that add features to Kubernetes applications. E.g. Aporeto
Service Partners	Consulting or management services to help companies implement Kubernetes in commercial applications. E.g. NebulaWorks

Power vs Interest Grid

Figure 1 shows the Power vs Interest grid, which divides the stakeholders in four categories:

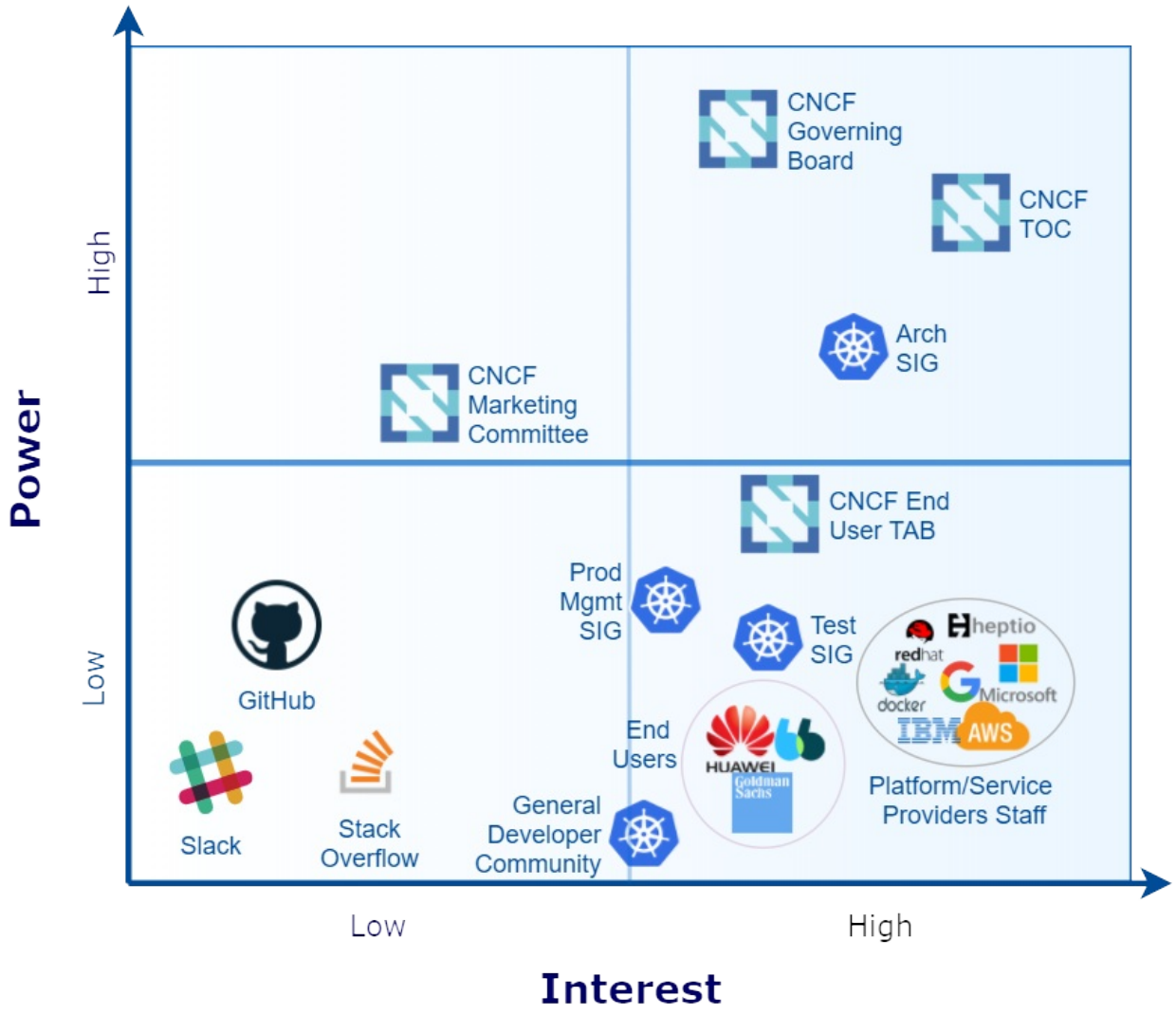


Figure 1. Power vs Interest Grid

- **High power and high interest** : Given that the CNCF Governing Board manages budget matters there is no doubt it has the maximum power on its projects, followed by the CNCF TOC, with the attribution to add and remove projects such as Kubernetes. Considering Kubernetes exclusively, the Architecture SIG is key because it decides the future of the project.
- **High power and low interest** : The CNCF Marketing Committee, derived from the CNCF Governing Board, takes care of the project brands and other business-related activities.
- **Low power and high interest** : Compared to the stakeholders with high power over Kubernetes, the CNCF End User TAB, platform/service providers staff, Test SIG, and Kubernetes general developer community depend on the continuous development of the project, what involves a high interest.
- **Low power and low interest** : Coordination platforms (e.g. GitHub) have little power and interest over the project given that other platforms could serve a similar purpose if they were not taken into account.

Analysis of Issues and PRs

After analyzing recent issues and Pull Requests (PRs), we obtained useful insight about Kubernetes project. For instance, Kubernetes already has a sophisticated method on working with issues and PRs: it uses templates, labels according problem domain and size, automated code checking tools, or bots for tracking. Moreover, SIGs elect their leaders democratically, while holding responsibility and autonomy. Every contributor must follow canonical principles like design security from ground-up or do unit tests. However, this complexity introduces a steep learning curve for new contributors in spite of existing documentation.

Integrators

Kubernetes project has a well-defined procedure to modify its code. New code is merged only if **approvers** agree with the modifications. Consequently, **approvers** are the **integrators**. The main Kubernetes repository integrators include [7]:

- bgrant0607
- brendandburns
- dchen1107
- jbeda
- monopole

Contacts

Table 2. Contact list

SIG	Contacts	Why?
Architecture	bgrant0607, jdumars	They are the Kubernetes Architects.
Contributor Experience	Phillels, parispittman	Their main purpose is to keep the contributors "happy and productive".
-	Experienced contributors (e.g. nikhita)	They actively participate on communication channels (e.g. Slack). They enjoy sharing ideas and solve newcomers' doubts.

Functional View

Kubernetes services are organized into two primary functions: master and worker services. Figure 2 shows a high-level diagram of Kubernetes Functional View.

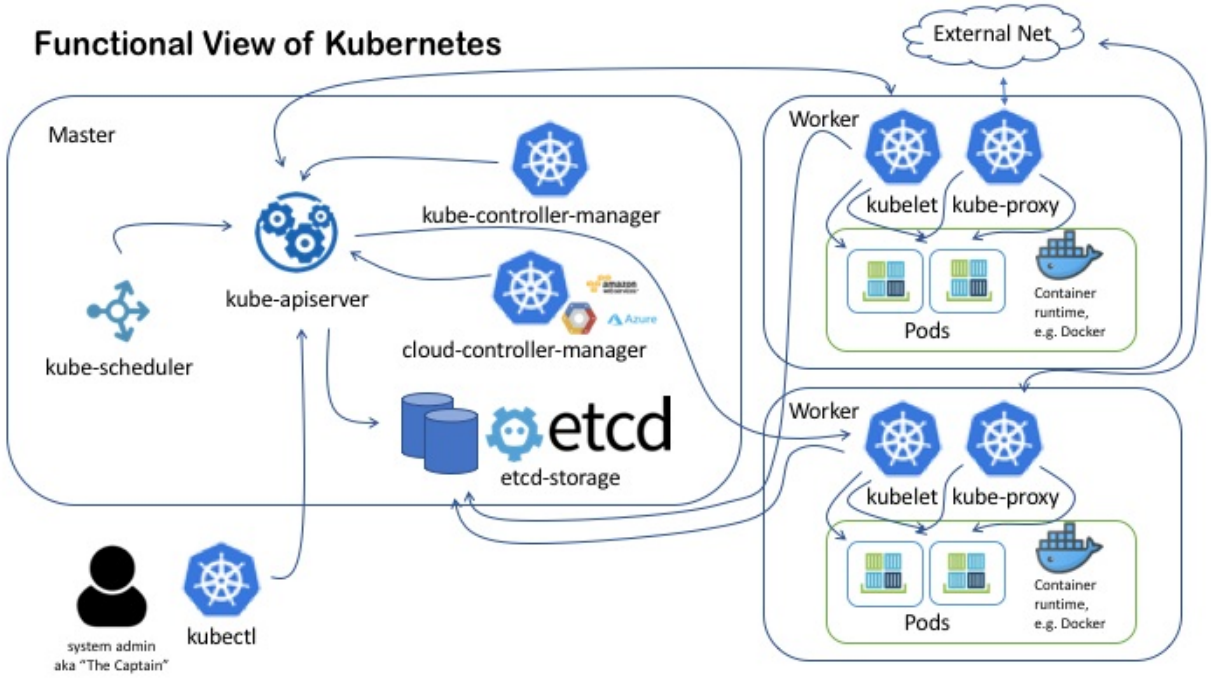


Figure 2. Kubernetes Functional View

This subsection is derived from the official Kubernetes Components document [8].

Master Node Components

The master node is responsible for managing the whole Kubernetes cluster. It provides the main interface for the user to interact with the system. Conceptually, there should be no container service running on the master node.

kube-apiserver

The `kube-apiserver` provides REST API service to interact with other components of the system. It processes every REST API request, validates it, and finally instructs the appropriate service to execute it. It employs `etcd-storage` as its persistent storage component.

etcd-storage

The master node uses etcd to store its states, for example jobs being scheduled, created and deployed; pod details and state; namespace and replication information; and so on.

kube-scheduler

The scheduler is responsible for finding the most suitable node for a pod or service to run, based on some constraints. By default, it tries to balance out the resource utilization of nodes [9]. It also watches resources available on all members of the cluster.

controller-manager

The controller manager is a process that encapsulates various controllers. In Kubernetes, a controller is defined as a control loop that watches the shared state of the cluster through the `kube-apiserver` and makes necessary changes to move the current state towards the desired state. It is organized into 2 modules: Kubernetes and cloud controller-manager.

kubectl

While technically `kubectl` is not a part of the master node, it is worth mentioning here as it is used to interact with the master node. `kubectl` is a command line interface for running commands against Kubernetes clusters. It communicates with the `kube-apiserver` via REST API service.

Worker Node Components

A Kubernetes worker node contains three main elements to be able to run appropriate containers inside the worker nodes. The elements are: `kubelet`, `kube-proxy`, and container runtime.

kubelet

`kubelet` is a service responsible for communicating with the master node and interacts with the container runtime service to run a Kubernetes pod. It gets the descriptive configuration of a pod from the `kube-apiserver` via REST API, then instructs the container runtime service to run the appropriate containers.

kube-proxy

`kube-proxy` is responsible for manageing network services in worker node, e.g. the exposure of network ports to the outside world, load balancing inbound traffic.

Container runtime

The container runtime is the software that is responsible for running containers. Kubernetes supports several runtimes: Docker, rkt, runc and any OCI runtime-spec implementation.

Context View

In this section, the scope of Kubernetes and its interaction with its surrounding environments are explained. The section consists of what the system does and does not do and also external entities that the system interacts with.

System Scope

Kubernetes is a container orchestration tool which is a system for managing containerized application [10]. The main tasks of Kubernetes are **deployment**, **scaling**, and **management** of containerized application. Each role is defined as follows.

- **Deployment**: manages the deployment of application by assigning application instances onto Nodes in a cluster
- **Scaling**: scale the application to keep up with user demand by adjusting cluster size and number of pod replicas
- **Management**: provide interface for managing the cluster and its containerized application

External Entities

The relationship between Kubernetes and its external entities are summarized in Figure 3:

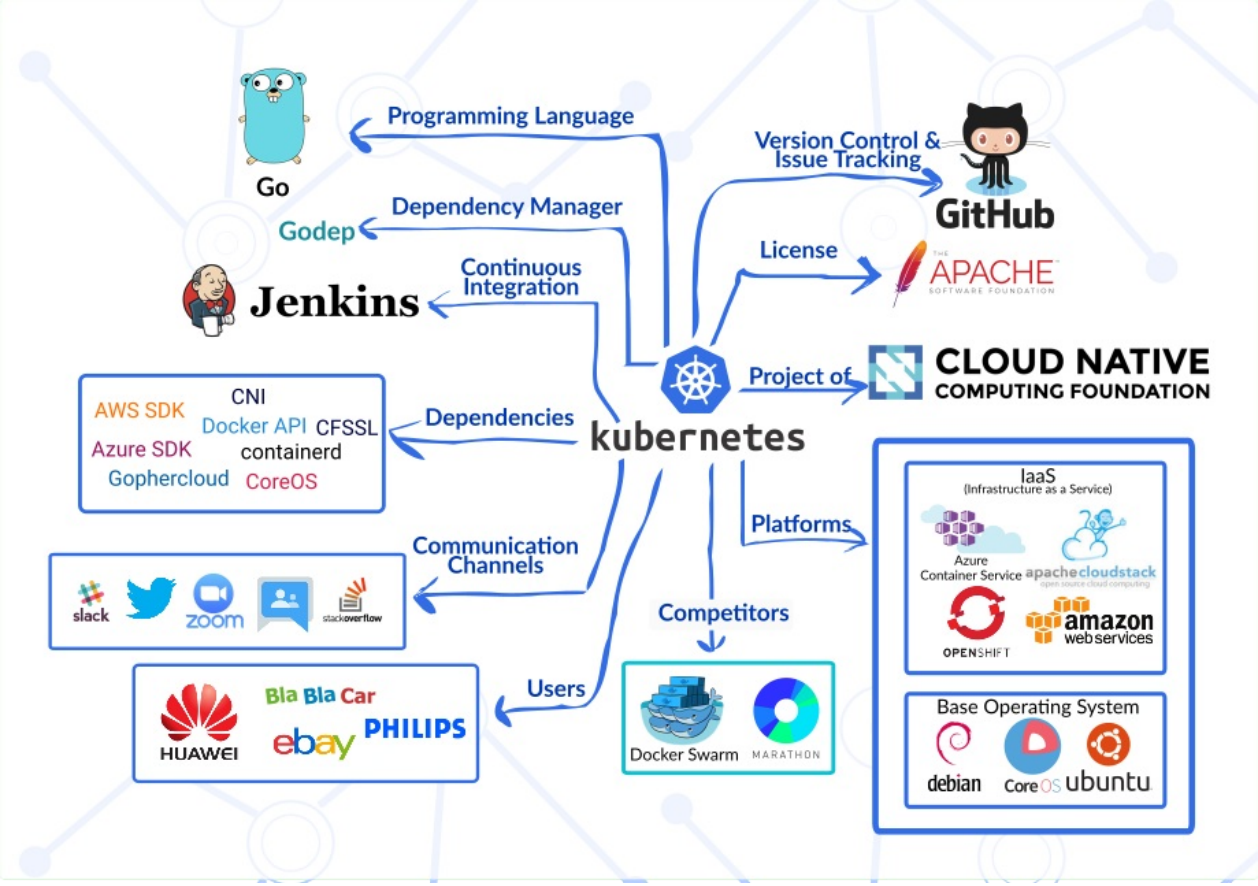


Figure 3. Context Model

We will discuss some of the entities regarding their relation with four focuses of the system: stakeholders, development, platforms, and competitors.

Stakeholders

In relation to the stakeholders discussed in the previous section, the context model includes interfaces between the Kubernetes system and some stakeholders. In this case, there are stakeholders entities: **CNCF** as the supervisor of the project which was listed as the acquirer, **GitHub** as a supplier, and some companies who uses Kubernetes in their system such as **Huawei**, **BlaBlaCar**, **eBay**, and **Philips** [5].

Development

The **Go** programming language is the main language used in Kubernetes development. Also, Kubernetes depends on several external libraries such as the **AWS SDK**, **Container Network Interface (CNI)**, **Docker API**, **Azure SDK**, and **Gophercloud**. Additionaly, Kubernetes is developed using **GitHub** for version controling and **Jenkins** for managing continuous integration.

Platforms

Kubernetes acts as a portable cloud platform and can run on various platforms [11]. Their documentation provides some extensive list of Infrastructure-as-a-Service (IaaS) providers that Kubernetes is compatible with. The list includes **Azure Container Service**, **Amazon Web Service (AWS)**, and **OpenShift**. Kubernetes communicates with those platforms with their respective SDK.

Competitors

Two of notably known competitors who also serve as a container orchestration tool are the **Docker Swarm** and Mesosphere's **Marathon**. **Marathon** is the orchestration platform for Apache Mesos, while Docker Swarm (now integrated into Docker system as the *Swarm Mode*) is the native clustering for orchestrating Docker containers.

Development View

Module Organization

Kubernetes source code is organized into layers of modules based on its functionality. The general overview of the partition of the modules is shown in Figure 4.

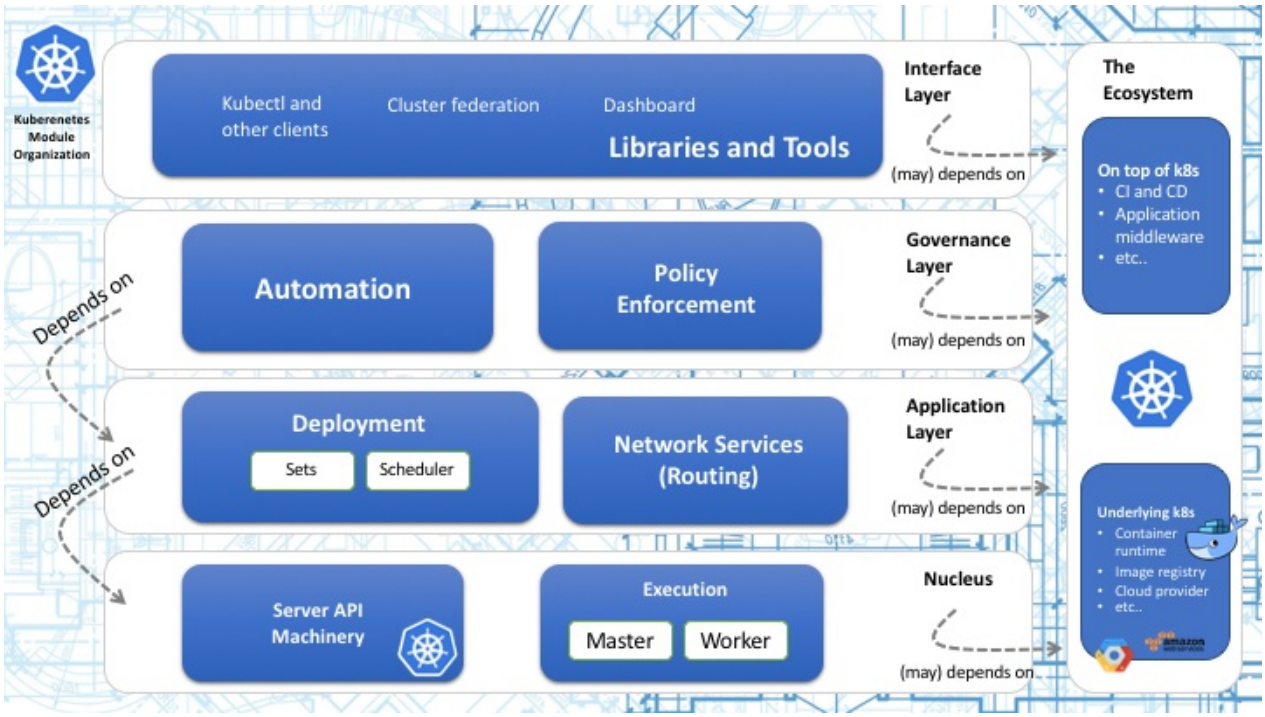


Figure 4. Kubernetes Module Organization

The layers and their functions description in this subsection are derived from the architectural roadmap of Kubernetes [12].

The Nucleus: API and Execution

The nucleus contains the minimum set of features needed to build the higher layers of the system and is composed of API and Execution modules. API modules provide a collection of REST APIs and the execution modules of Kubernetes are responsible for executing application within containers. The most important execution module in Kubernetes is `Kubelet`.

The Application Layer: Deployment and Routing

The application layer provides self-healing, scaling, application lifecycle management, service discovery, load balancing, and routing. Further, the deployment module provides container-centric methods and lifecycle controllers to support orchestration, while the routing module provides scheduling service.

The Governance Layer: Automation and Policy Enforcement

This layer contains the automation module which provides automatic scaling of the cluster and automatic provisioning of nodes. The policy enforcement module provides means to configure and discover default policies for the cluster.

The Interface Layer: Libraries and Tools

This layer contains commonly used libraries, tools, systems, and UIs in Kubernetes project. One of the most important tools in this layer is `kubectl`. It also contains other client libraries such as client-go and client-python.

The Ecosystem

This layer is not really part of Kubernetes, but it provides functionality commonly needed to deploy and manage containerized applications. The most popular example of this supporting layer is Docker, which runs the actual container in nodes. Modules used to communicate with cloud provider also belong here.

Common Design Model

In this section, we will look at the common design models that Kubernetes developers can refer to. We looked at the common processing, design standardization, and testing standardization that Kubernetes developer teams have. According to [4], these common design models can be used so that the system is easy to understand, operate, and maintain .

Common Processing

In the development of Kubernetes, there is a common processing across different elements in order to simplify cross-element integration of code units. Two of the most important common processing elements are instrumentation and logging. They are part of the Kubernetes developer guide [13].

Through instrumentation, we can measure the performance of a certain metric. The ability of a system to do instrumentation can help developers during the development. Kubernetes follows the general Instrumentation advice from the Prometheus Go client library [14]. According to the Kubernetes's instrumentation guide [15], Kubernetes added an extra convention on instrumentation, mainly the naming of the metrics. The naming of a metric should be as follow: `<<component_name>>_<metric>`. This is done to avoid collisions of metrics between the components since sometimes the measurement of one metric is done in several components.

The other way to understand the performance and real way of working of our system is by logging. Logging is one of the common design models that are essential for developers during development to ensure that the system worked as intended and can be understood by another developer. Logging is also useful during debugging to know the error encountered during the build. According to the logging convention [16], Kubernetes uses the built-in glog logging function in Golang. The logs provided by glog are standard, such as error log, warning log, and info log.

Design Standardization

To ensure a project to be maintainable, cohesive, and easy to reproduce, developers that are working on the project need to adhere to a certain design standard. Since Kubernetes has a variety of developers who intend to contribute to the project, the Kubernetes team formulated a design standardization for the coding, which is stated in the Coding Conventions document [17]. Since Kubernetes is developed using Go language, the main coding conventions that Kubernetes developers have to adhere is the Effective Go guideline [18]. Developers also have to avoid Go landmines to ensure the effectiveness of the code [19]. Several other code conventions include code commenting, command-line flags, naming, and other conventions, such as API and logging conventions [16][20].

Codeline Model

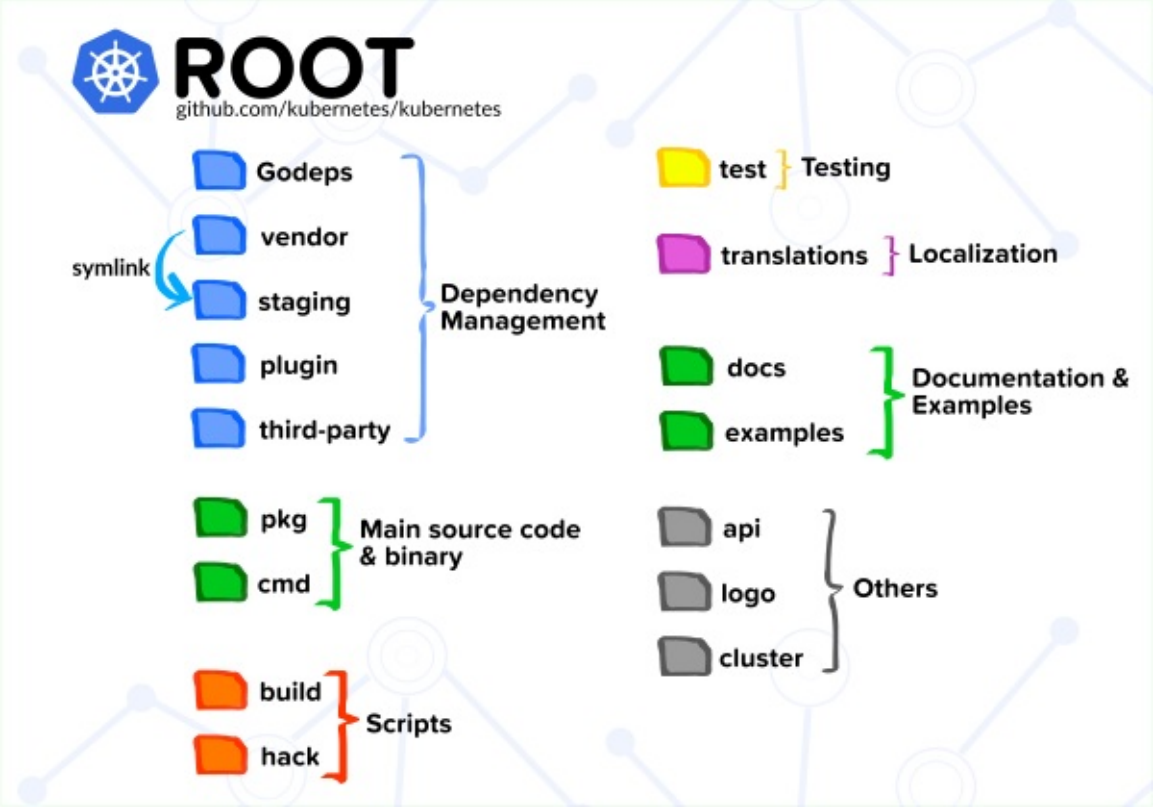


Figure 5. Source Code Structure

The overall Kubernetes project's source code structure is depicted in Figure 5. Kubernetes is developed on top of packages. The main packages are organized inside the `pkg` folder, while dependencies on the external package are managed by Godep.

The description of each category illustrated in Figure 5 is covered in more detail in Table 3.

Table 3. Source Code Structure

Category	Description
Dependency Management	This category mainly consists of folders generated by Godep. The `vendor` folder also contains packages from Kubernetes external repository. These packages are symlinked to the `staging` folder for development purposes.
Main Source Code & Binary	The `pkg` folder is where all source code is organized in multiple packages. There is also the `cmd` folder containing binaries (source code for main executables).
Scripts	Scripts for automating build process are stored in `build` folder, while `hack` folder contains shell (`.sh`) scripts for development automation.
Testing	For unit and integration testing purposes.
Localization	Contains translations to multiple languages for messages shown by `kubectl` .
Documentation & Examples	Contains documentation for `kubectl` , API reference, and some examples of how to run real applications in Kubernetes.
Others	This category includes miscelaneous stuff such as API specification, logo images, and the deprecated `cluster` folder.

Additionally, Table 4 lists the code structure inside the `cmd` , describing how main executables source code is organized. The related package (from the `pkg` folder) used by the respective executables is also listed in the table.

Table 4. Folders under `cmd`

Directory	SIG Owner	Description	Related Package
`clicheck`		common CLI checking	`pkg/kubectl`
`cloud-controller-manager`	sig-api-machinery	External controller manager for running cloud specific controller loops	`pkg/cloudprovider`
`hyperkube`	sig-release	An executable that can morph into other kubernetes binaries	`pkg/kubectl`
`kube-apiserver`	sig-api-machinery	Main api server and master for the cluster	`pkg/kubeapiserver`
`kube-controller-manager`		Monitoring replication controllers	`pkg/cloudprovider` , `pkg/controller`
`kube-proxy`		For managing proxy server between localhost and API server	`pkg/apis` , `pkg/kubelet` , `pkg/proxy` , `pkg/master`
`kube-scheduler`	sig-scheduling	Scheduler for assigning nodes to pods	`pkg/scheduler`
`kubeadm`	sig-cluster-lifecycle	Toolkit to setup Kubernetes cluster	`pkg/api`
`kubectl`	sig-cli	Main CLI for running commands against clusters	`pkg/kubectl`
`kubelet`	sig-node	Maintaining a set of containers on a particular host VM.	`pkg/kubelet`
`kubemark`	sig-scalability	Performance testing tool	`pkg/kubemark`

Evolution Perspective

Since the first commit at GitHub on June 7, 2014, Kubernetes has evolved throughout the years, having a lot of new feature being added to the system and bug being fixed. In this section, we look at the evolution of the Kubernetes software throughout its development.

Kubernetes project was first developed by Google. The project was heavily influenced by Borg, a similar project at Google, and was made open-source [21] [22]. Kubernetes follows the semantic versioning convention [23] when releasing an update, that is Kubernetes `vX.Y.Z` where `X` denotes a major release, `Y` denotes a minor release, and `Z` denotes a patch release.

Before the first major release of Kubernetes `v1.0.0`, the project was still not ready for use in production. The project was released on a bi-weekly basis up until `v0.21.0`, which was released on July 8, 2015, and every minor release has new features being added, such as AWS support in the `v0.5.0` release and Kubernetes UI with dashboard components in the `v0.16.0` release.

When the first major `v1.0.0` of Kubernetes was released in July 13, 2015, Google ceded the control over the project to Cloud Native Computing Foundation (CNCF) and the software is considered ready for use in production environment [24]. After this major release, the project was released on a quarterly basis every year. Some of the most notable releases were [25]:

1. `v1.2.0`: The introduction of Special Interest Group (SIG) in the project and the scaling improvements where at that version, Kubernetes could support up to 1000 nodes.
2. `v1.5.0`: Simplification of cluster deployment and Windows Server Container support.
3. `v1.6.0`: Migration to etcd v3 which can support up to 5000 nodes and overall improved documentation of the project through the Community Repository.
4. `v1.7.0`: Security enhancement such as node authorizer and TLS certificate rotation; and added internationalization support for the Chinese and Japanese language.
5. `v1.8.0`: Improved workload API to support Apache Spark.

The most recent version of Kubernetes, up to the time of writing, is the `v1.10.0` release. Besides the new features and bug fixes on each release, Kubernetes also improves its documentation and its product management through separate repositories such as Community, SIG-Release, and Features so that the development of Kubernetes is more organized.

Technical Debt

In this section, we look at all possible sources of technical debt in Kubernetes project. We have performed several types of analysis, which are the analysis of general technical debt as well as specific testing debt and historical analysis.

Manual Identification

As part of the analysis of Kubernetes technical debt, we manually skimmed the Kubernetes's GitHub repository and the code structure looking for technical debt traces. We found that Kubernetes community members are familiar with the terminology and are aware of some areas concerning this technical debt as there are issues specifically labelled as `kind/technical-debt` such as scheduling (58346, 56236) and api-machinery (54511, 44263) issues. These issues date as far back as 2015, which means that technical debt exists and the project maintainers make efforts to manage and deal with technical debt in one way or another. Besides that, a quick search through the code reveals many TODO comments regarding work that is still required inside several source code files, some of them as important as kubelet.go, master.go, gce.go, among others. It is worth noticing that many of these files also under the supervision of api-machinery SIG.

Automated Code Quality Analysis

Here we analyze Kubernetes project regarding its technical debt using available tools that examine the quality of the code. For that purpose, we used Codebeat as a tool to assess code quality automatically. This tool inspects code quality by calculating a set of metrics that are related to software quality, extensibility, and maintainability [26].

Codebeat assesses metrics in the code's function and namespace level. It analyzed the Total Complexity and Code Duplication aspects of the code, among others.

Total Complexity

The high number of complexity indicates that the code contains too much logic and should probably be broken down into smaller elements. The complexity of the code can be measured using several metrics such as the number of arguments and return values in a function; and the number of conditional cases. One of the example of this can be seen in Figure 6. In this example, there are eight arguments in the function `AttachContainer` which might cause confusion when using this function.

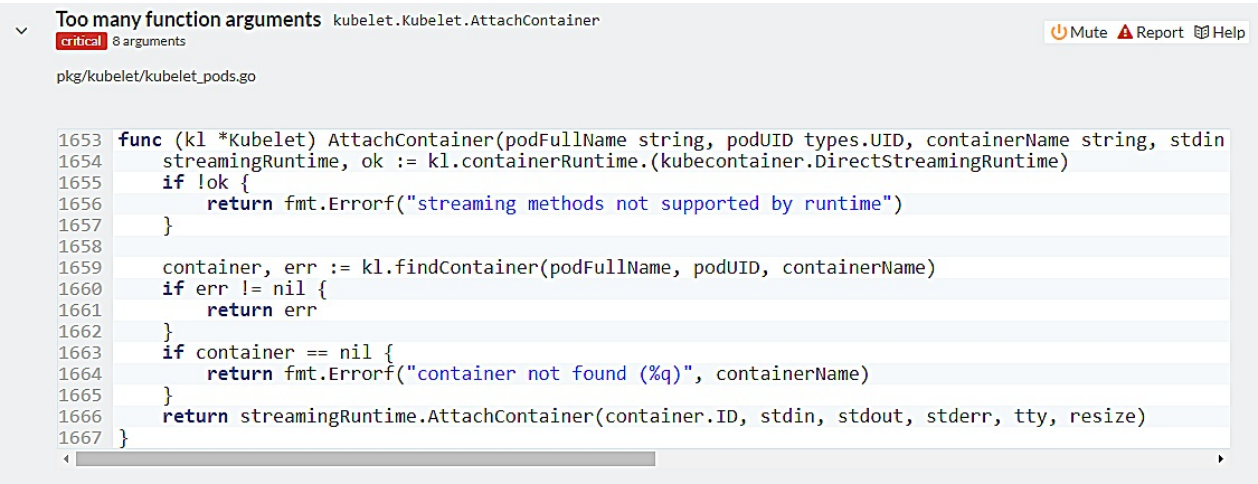


Figure 6. Complexity code example

Code Duplication

One of the canonical principles in software development is *Don't Repeat Yourself*: "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system" [27]. From that perspective, the number of duplication in the code is considered as one of the most important metrics when measuring the code quality as it may lead to difficulty during maintenance since changing one instance of code would need replication in other instances as well.

One example of code duplication pointed out by Codebeat is in `pkg/kubelet/apis/cri/runtime/v1alpha2/api.pb.go`. The developer used the same lines of code in several functions. This means that they have to change all code in corresponding functions whenever they need to change logic.

Testing Debt

In this section, we will discuss some technical debt related to testing that is found in the Kubernetes system. The analysis is based on the Testing Guide, Unit Test Coverage, and GitHub Issues of their main repository [28][29]. Overall, the Kubernetes community puts a lot of effort in managing testing debt because they are aware that the project is enormous and involve many contributors, so testing debt is unavoidable, but they need to manage it somehow.

Unit Test Coverage

We did an analysis on unit test coverage of Kubernetes by using script provided by Kubernetes: `make test KUBE_COVER=y` [28]. We found that the average coverage is **52%** among 1.833 source files. As we can see in Figure 7, there are still many files which have below 10% coverage; some has even 0% coverage. Although the testing guide provided is quite comprehensive, we can see that there are still some debts left of writing tests for some source files.

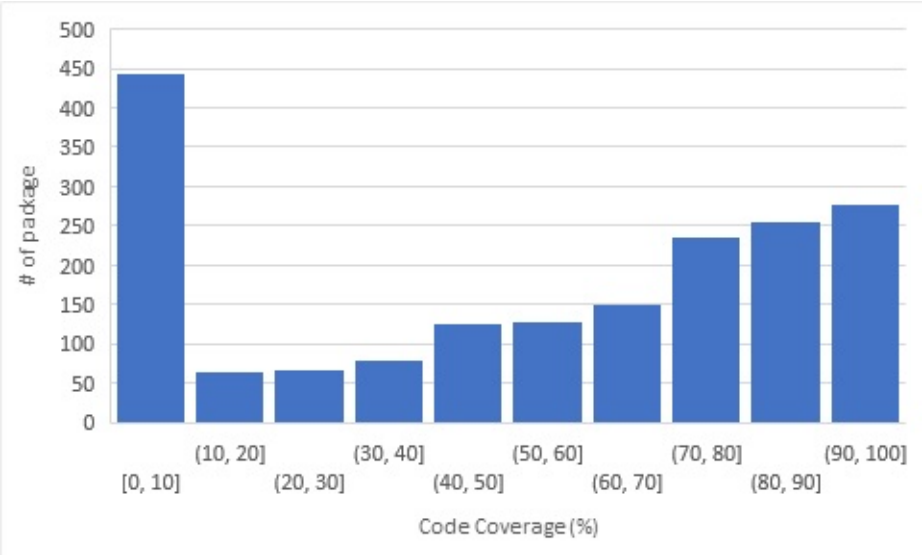


Figure 7. Distribution of Code Coverage

Nonetheless, this just shows that there are lines of code that are not covered in unit tests. As mentioned before, Kubernetes does incorporate integration and E2E test suite, so this percentage is just indicator for coverage by unit testing.

Flaky Tests

In the Kubernetes contribution guide, there is a special testing-related section about Flaky Tests [30]. The flaky test is a test that occasionally fails. Because the nature of Kubernetes system in the real world could be run in various environments with resource limitations, the team tries to avoid flaky test as much as possible. They encourage contributors to run tests multiple times to reduce the chance that it is not flaky. They even have a dedicated issue label `kind/flake` for contributors to report flaky tests. As of now, there are 20 open issue related to flake tests, and if they are not dealt with properly, it can be a technical debt that they have to address later when the failed test cause a queue of PRs to be merged.

Historical Analysis

Scope Definition

Considering the vast amount of Kubernetes objects (at least 22.286 unique objects in the last nine major releases!), we decide to limit the context of this analysis to top 20 most important objects in Kubernetes. Further, we also define the number of commits to each object as the indicator of how important an object is, i.e. higher commit amounts indicates more importance. Table 5 shows the top 20 objects (without test object) in terms of commits amount.

Table 5. Top 20 objects based on commit amount.

Path	Commits
kubelet/kubelet.go	1734
master/master.go	643
kubelet/rkt/rkt.go	343
kubectl/cmd/util/factory.go	333
cloudprovider/providers/aws/aws.go	332
kubectl/cmd/cmd.go	249
cloudprovider/providers/gce/gce.go	231
controller/deployment/deployment_controller.go	229
proxy/iptables/proxier.go	206
kubectl/cmd/run.go	205
api/testing/fuzzer.go	194
apis/componentconfig/types.go	193
volume/glusterfs/glusterfs.go	172
kubelet/container/runtime.go	165
printers/internalversion/describe.go	162
kubectl/cmd/util/helpers.go	160
kubelet/kubelet_pods.go	153
controller/controller_utils.go	148
apis/extensions/types.go	147
kubectl/cmd/create.go	147

We also define the temporal scope of the major releases, which are:

1. Kubernetes v1.2.7,
2. Kubernetes v1.3.10,
3. Kubernetes v1.4.4,
4. Kubernetes v1.5.0,
5. Kubernetes v1.6.0,
6. Kubernetes v1.7.0,
7. Kubernetes v1.8.0,
8. Kubernetes v1.9.0, and
9. Kubernetes v1.10.0-beta4.

They represent the nine latest major releases of Kubernetes.

Evolution Matrix Analysis

We start our analysis by building an evolution matrix [31] to inspect pattern in the length of codes in those 20 objects. The metric used to build this evolution matrix is the line of code of each object including comments. The width and height metric of the rectangle are defined to be the same, i.e. the lines of code.

Figure 8 exhibits the evolution matrix of the 20 most committed objects. Note that the 20 most committed objects start to exist from version 1.5.0. To see the complete version, interested readers can refer to our version of complete evolution matrix of Kubernetes for nine latest major releases.

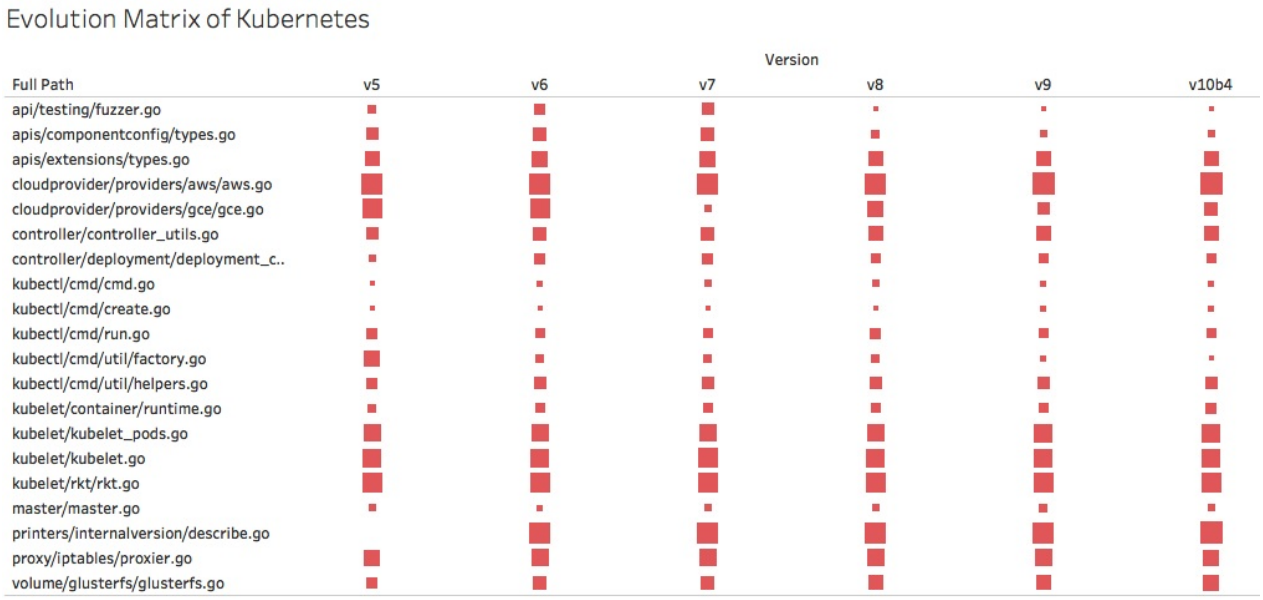


Figure 8. Evolution matrix of the top 20 objects.

From the evolution matrix of the 20 most committed objects, we can observe that in general the number of lines is increasing but not too significant relative to time. In terms of artefacts described in [31], these stable objects can be categorized as persistent. This observation may indicate the maturity of these objects and also the engineering process in this project.

Beside of the persistent objects, we also see the pattern of a pulsar in object `cloudprovider/providers/gce/gce.go`. Pulsar is a term for an object that grows and shrinks as time progresses because of repeated modifications. Pulsar objects can be seen as hotspots in the system: for every new version of the system, changes on a pulsar object must be performed [31]. In this specific case, the object `gce` is responsible to provide methods needed for Kubernetes to be deployed in a Google Compute Engine environment. Reducing hotspots in this object is a form of technical debt.

Another thing that we can inspect is that there is also a pattern of a white dwarf in object `kubectl/cmd/util/factory.go`. In version 1.5.0, this object had 1,334 line of codes, and in the following version of 1.6.0, the number of lines was reduced drastically to 506 line of codes. It keeps shrinking through time until it gets to its current size of 337 lines in 1.10.0-beta4. From the blame view, we can observe that the methods in this object were refactored to various objects in `pkg/kubectl/cmd/util/`. This indicates that this object was already identified as a technical debt and refactored accordingly.

To conclude, we can observe that most of the objects in Kubernetes are persistent objects. This fact is a proof that Kubernetes developers team have done a remarkable job in designing objects and enforcing strict engineering policies.

Conclusion

Throughout the whole chapter, we have analyzed Kubernetes' software architecture by looking at its stakeholders, viewpoints, and perspectives.

Stakeholders analysis provides us insight into how Kubernetes developers are organized through SIGs and how feature development or bug fixing are being performed on GitHub. Looking into the functional and context views of Kubernetes, we studied how Kubernetes is organized as a system, how the components interact with each other, and also how Kubernetes interacts with external parties.

Furthermore, we have analyzed the development view to gain insights on how the development of Kubernetes progresses, especially when there are lots of developers working on the same project. We learned how conventions, such as coding convention, helped developers on the previously mentioned problem. Looking at the evolution of Kubernetes, we learned about how Kubernetes has matured throughout the years.

Finally, we thoroughly analyzed the technical debt that Kubernetes has and how it is managed. Even though there are still technical debts to be solved, we firmly believe that Kubernetes will keep growing due to its active community and reliable architecture.

References

[1] “What Is Kubernetes?” Kubernetes. Accessed April 5, 2018. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[2] Vaughan-Nichols, Steven J. “ Google Releases Kubernetes 1.0: Container Management Will Never Be the Same.” ZDNet. Accessed April 5, 2018. https://www.zdnet.com/article/google-releases-kubernetes-1-0/.

[3] Conway, Sarah. “Kubernetes Is First CNCF Project To Graduate.” Cloud Native Computing Foundation (blog), March 6, 2018. https://www.cncf.io/blog/2018/03/06/kubernetes-first-cncf-project-graduate/.

[4] Rozanski, Nick, and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012.

[5] “Case Studies.” Kubernetes. Accessed April 5, 2018. https://kubernetes.io/case-studies/.

[6] “Partners.” Kubernetes. Accessed April 5, 2018. https://kubernetes.io/partners/.

[7] Kubernetes: Production-Grade Container Scheduling and Management. Go. 2014. Reprint, Kubernetes, 2018. https://github.com/kubernetes/kubernetes.

[8] “Kubernetes Components.” Kubernetes. Accessed April 5, 2018. https://kubernetes.io/docs/concepts/overview/components/.

[9] bobsky. “Advanced Scheduling in Kubernetes.” Accessed April 5, 2018. https://kubernetes.io/blog/2017/03/advanced-scheduling-in-kubernetes.

[10] “Kubernetes Documentation.” Kubernetes. Accessed April 5, 2018. https://kubernetes.io/docs/home/.

[11] “Picking the Right Solution.” Kubernetes. Accessed April 5, 2018. https://kubernetes.io/docs/setup/pick-right-solution/.

[12] Kubernetes Architectural Roadmap. Go. 2016. Reprint, Kubernetes, 2018. https://github.com/kubernetes/community/blob/master/contributors/devel/architectural-roadmap.md.

[13] Kubernetes Developer Guide. Go. 2016. Reprint, Kubernetes, 2018. https://github.com/kubernetes/community/tree/master/contributors/devel.

[14] “Instrumentation | Prometheus.” Accessed April 5, 2018. https://prometheus.io/docs/practices/instrumentation/.

[15] Intrumenting Kubernetes. Go. 2016. Reprint, Kubernetes, 2018. https://github.com/kubernetes/community/blob/master/contributors/devel/instrumentation.md.

[16] Kubernetes Logging Conventions. Go. 2016. Reprint, Kubernetes, 2018. https://github.com/kubernetes/community/blob/master/contributors/devel/logging.md.

[17] Kubernetes Coding Conventions. Go. 2016. Reprint, Kubernetes, 2018. https://github.com/kubernetes/community/blob/master/contributors/guide/coding-conventions.md.

[18] “Effective Go - The Go Programming Language.” Accessed April 5, 2018. https://golang.org/doc/effective_go.html.

[19] “Golang Landmines.” Gist. Accessed April 5, 2018. https://gist.github.com/lavalamp/4bd23295a9f32706a48f.

[20] Kubernetes API Conventions. Go. 2016. Reprint, Kubernetes, 2018. https://github.com/kubernetes/community/blob/master/contributors/devel/api-conventions.md.

[21] “Google Open Sources Its Secret Weapon in Cloud Computing | WIRED.” Accessed April 3, 2018. https://www.wired.com/2014/06/google-kubernetes/.

[22] Oppenheimer, David. “Borg: The Predecessor to Kubernetes.” Accessed April 3, 2018. https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes.

[23] Preston-Werner, Tom. “Semantic Versioning 2.0.0.” Semantic Versioning. Accessed April 3, 2018. https://semver.org/.

[24] “As Kubernetes Hits 1.0, Google Donates Technology To Newly Formed Cloud Native Computing Foundation.” TechCrunch (blog), July 21, 2015. http://social.techcrunch.com/2015/07/21/as-kubernetes-hits-1-0-google-donates-technology-to-newly-formed-cloud-native-computing-foundation-with-ibm-intel-twitter-and-others/.

[25] “Kubernetes.” Accessed April 3, 2018. http://blog.kubernetes.io/.

[26] “Function-Level Metrics.” Accessed April 5, 2018. https://hub.codebeat.co/docs/software-quality-metrics.

[27] “Dont Repeat Yourself.” Accessed April 5, 2018. http://wiki.c2.com/?DontRepeatYourself.

[28] Kubernetes Testing Guide. Go. 2016. Reprint, Kubernetes, 2018. https://github.com/kubernetes/community/blob/master/contributors/devel/testing.md.

[29] Kubernetes Issues. Go. 2014. Reprint, Kubernetes, 2018. https://github.com/kubernetes/kubernetes/issues.

[30] Kubernetes Flaky Tests. Go. 2016. Reprint, Kubernetes, 2018. https://github.com/kubernetes/community/blob/master/contributors/devel/flaky-tests.md.

[31] Lanza, Michelle, and Stéphane Ducasse. “Understanding Software Evolution Using a Combination of Software Visualization and Software Metrics.” L’objet 8, no. 1–2 (June 30, 2002): 135–49. https://doi.org/10.3166/objet.8.1-2.135-149. re Visualization and Software Metrics.” L’objet 8, no. 1–2 (June 30, 2002): 135–49. https://doi.org/10.3166/objet.8.1-2.135-149.

Lighthouse

By Kanav Anand, Nouri Khalass, Ernst Mulders, and Michiel van Spaendocnk from TU Delft

Table of Contents

Introduction

Lighthouse is a tool that allows web developers to evaluate their website according to modern standards. It analyzes web apps and web pages and collects performance metrics and insights on developer best practices. The project was released by Google as an open source initiative to improve the quality of websites.

Stakeholders

In order to determine the stakeholders of the Lighthouse project, we've looked at the contribution history on Github as well as their demo video of the Google I/O of last year. The most important group of people are the so called Lighthouse keepers. These are the top contributors of the project, and work for Google.

Rozanski and Woods classifications

According to the classification scheme by Rozanski and Woods the Lighthouse stakeholders can be grouped as follows.

Acquirers: *Oversee the procurement of the system or product*
That would be Google as owner of the project. Google invests the money that pays the salaries of the top contributors whom work for Google.

Assessors: *Oversee the system's conformance to standards and legal regulation*
Lighthouse keepers, although they quite surely have a legal team behind them from Google.

Communicators: *Explains the system to other stakeholders via its documentation and training materials*
Eric Bidelman and Brendan Kenny were on stage on the I/O conference talking about and explaining Lighthouse. If we look at the readme.md there are 44 contributors. Paul Irish, Patrick Hulce, and Sam Saccone were among the top of these 44.

Developers: *Construct and deploy the system from specifications (or lead the teams that do this)*
The Lighthouse keepers as well as outside contributors.

Maintainers: *Manage the evolution of the system once it is operational*
The Lighthouse keepers, and in particular Paul Irish.

Suppliers: *Build and/or supply the hardware, software or infrastructure on which the system will run*
This is everyone that uses the plugin in their Chrome browser, or clones the repository and runs it themselves.

Support staff: *Provide support to users for the product or system when it is running*
The Lighthouse keepers are keeping track of many of the open issues, and helping the people that are creating them. Also, the Lighthouse keepers are easily reached through Twitter.

System administrators: *Run the system once it has been deployed*
Same as suppliers, so the users themselves.

Testers: *Test the system to ensure that it is suitable for use*
The users and the Lighthouse keepers. Everybody that notices a problem while testing the tool can submit a bugreport.

Users: *Define the system's functionality and ultimately make use of it*
Webdevelopers.

Additional stakeholders

The classifications or Rozanski and Woods nearly cover all stakeholders. But there are more to come up with. For Lighthouse we could identify **beneficials**, the people who browse the web and enjoy the better website because the developers used Lighthouse. Furthermore we could identify **Investors**, these are the people (or companies) who have a financial interest. For Lighthouse these investors would be Google and thereby its mother company Alphabet.

Integrators

Integrators are the users who check the pull request. It's their task to keep the project stable and on track for the future roadmap. With lots of different contributors its hard to keep up with the quality standards. As described in How do project owners use pull requests on Github? there are a lot of different evaluation points to check before the pull request is merged within the master project. Interesting enough Lighthouse also provides a commandline based version, which in fact helps integrators by automatically checking if their web project is still up to the set quality standards after pulling the request branch. So the team is helping integrators with their tasks (at least for web based projects).

In order to find the integrators we've looked at the commits to the master branch. In these commits we see the original author of the code, as well as the integrator. It is easy to spot the people we identified as most influential (see 1.7 for the full list). Paul Irish and Patrick Hulce arise as the most prominent integrators. Together they've done all pull requests.

When looking at the pull request you see that Paul and Patrick often discuss the impact of the change for the majority of the users. If it its a worthy fix, and doesn't break the usage for the user they'll go through with it. In some cases, you'll see that they'll even accept the pull request whilst knowing it will brake the application of some, such as in this pull request.

Relevant people

By analyzing the online community around Lighthouse we found that the following people were the most involved in the project.

Name	Twitter	GitHub	Role
Paul Irish	@paul_irish	@paulirish	Google, Developer Tooling
Eric Bidelman	@ebidel	@ebidel	Google, Developer Relations
Brendan Kenny	@brendankenny	@brendankenny	Google, Developer Relations
Patrick Hulce	@patrickhulce	@patrickhulce	Google, Developer Tooling
Paul Lewis	@aerotwist	@paullewis	Google, Developer Relations
Sam Saccone	@samccone	@samccone	Android
Rob Dodson	@rob_dodson	@robdodson	Google, Developer Relations

Context View

The context view of the system describes the relationships, dependencies, and interactions between the system and its environment [1]. This section describes the system's scope and responsibilities as well as relations with its environment consisting of users and external entities.

System Scope and Responsibilities

Google created Google Lighthouse, as part of its effort to support progressive web apps, to establish standards for today’s web. It is an open-source auditing tool that helps developers improve the performance and accessibility of their web projects. It can be used by anyone for free to see how their website stacks up against Google’s high standards. The main responsibility of Google Lighthouse is to grade websites on the following criteria :

- Security: Is it being served from a secure origin?
- Accessibility: Does it work for all users?
- Perceptual Speed: Do users perceive it as fast? How your content loads is just as important as how fast it loads.
- Offline Connectivity: Will it load offline or in unreliable network conditions?

External Entities

There are several external entities surrounding Lighthouse's environment. We first enlist them below and display them later as a context diagram.

- The organization that made the starting efforts: Google.
- The license of the project: Apache Licence 2.0.
- Programming languages used in the project: NodeJS.
- Tools used for code quality assurance: JSDoc, Closure Compiler.
- Tool used for Development and Issue tracking: GitHub.
- Tool used for error reporting and logging: Sentry.js.
- Testing framework and coverage tools: Istanbul.js, Mocha.js.
- Tools used for packaging and shipping the system: YARN, npm.
- Tools used for Continuous Integration: Travis CI.
- Users of the system: companies or individuals that build web applications and are interested in improving the performance and accessibility of their web applications.
- The development community: open source community.

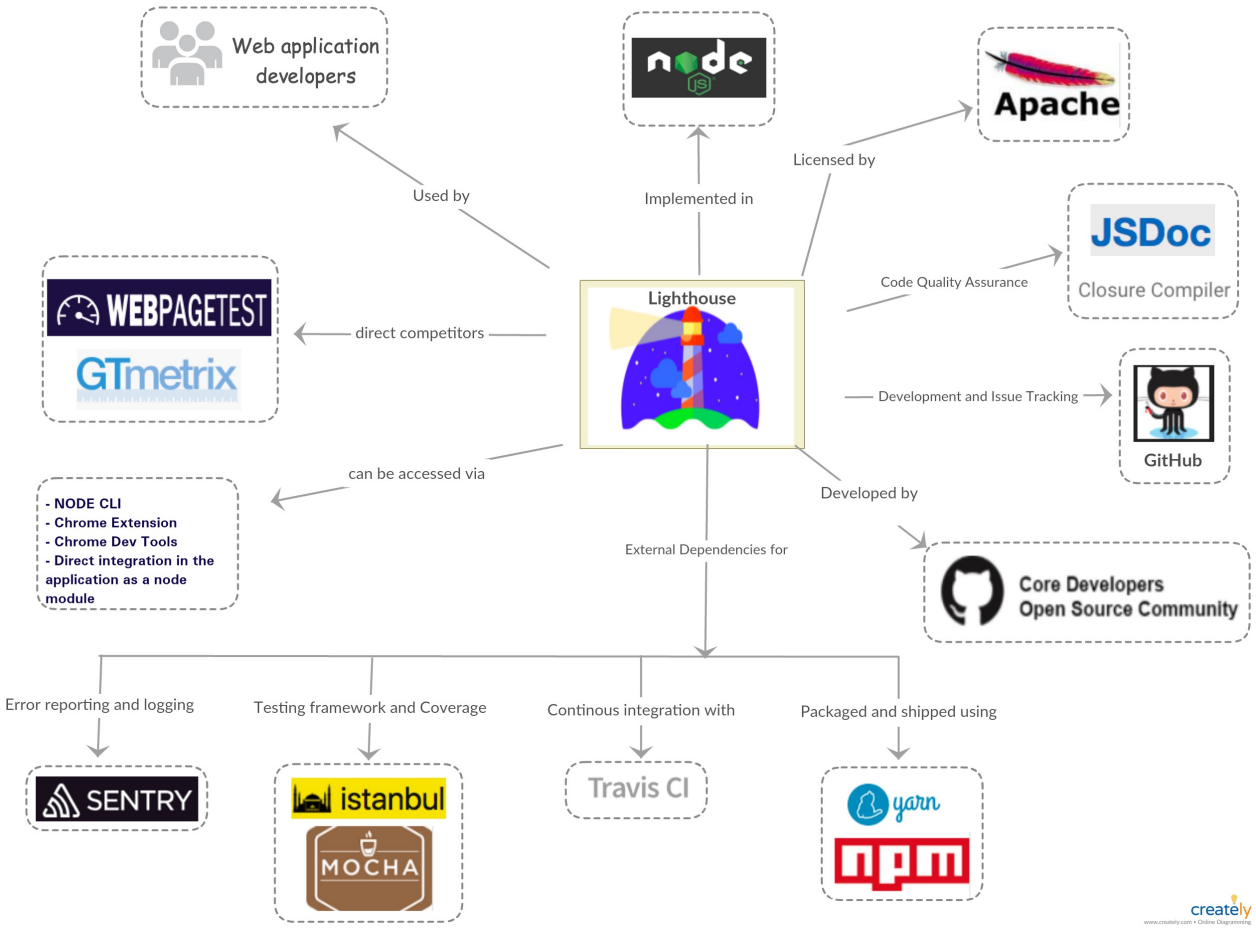


Figure 1: Context View for Lighthouse

Using Figure 1, we try to provide a brief overview of the environment of Lighthouse and how it interacts with these entities.

- **Lighthouse** Lighthouse is a tool used to test the quality of web applications. It was started by Google and is currently licensed under Apache Licence 2.0. It is a completely automated tool and requires only the URL of a web application.
- **Programming languages** Lighthouse is implemented using NodeJS and it requires the version to be Node 6 or higher.Along with YARN, npm is used as a dependency manager.
- **Quality assurance** Lighthouse puts a strong emphasis on the quality of the product as it is used for the sole purpose to test the quality of other web applications. Thus, multiple tools are used in order cover every nook and corner of the application. It depends on Istanbul.js and Mocha.js as its main testing frameworks. In addition to above-mentioned frameworks, Lighthouse also uses Sentry.js for reporting unexpected errors, that is, runtime exceptions.
- **Communication and Issue tracking** All the communications related to the development of lighthouse are done using GitHub. It follows a simple project board model with issues categorized under different projects. Apart from development communication, Lighthouse extensively uses Twitter to reach a wider audience for announcing the new release and other promotional events.
- **Continuous Integration** TravisCI is responsible for continuous integration of lighthouse. It is one of the key dependency of Lighthouse as being an open source project, it is very difficult to review every contribution effectively. Thus, TravisCI runs all the existing tests in the project on each pull request raised and it is merged only if it is backward compatible. It is necessary to ensure the stability of the system.
- **Users** Lighthouse is extensively used by companies or individuals that are in web applications development business. Many users include Lighthouse as a part of their CI to constantly monitor the changes affecting their web application quality.
- **Community** Lighthouse has a strong open source community consisting of active contributors. It has around 2687 forks on GitHub and with around 1800 closed issues.

Development View

The development view of a project describes the architecture that supports the software development process [2]. This section will address what modules Lighthouse is composed of, what kind of standardized practices take place and how the project is structured.

Modules

Lighthouse consists of multiple modules with some being dependant on others. A small list of all modules present in Lighthouse will be given after which the module structure will be explained more in depth.

Module Name	Description
`lighthouse-core`	Core part of Lighthouse which performs the tests and comes up with a report based on the results.
`lighthouse-cli`	Command-line front-end for Lighthouse that wraps `lighthouse-core` to be interfaceable via the command line.
`lighthouse-extension`	Packages `lighthouse-core` to be usable as a Google Chrome extension.
`lighthouse-logger`	Logger module used for information reporting to the console.
`lighthouse-viewer`	Application that visualizes Lighthouse reports.

The main part of Lighthouse is `lighthouse-core`. This is the part that performs the different audits and gives a score based on the success of those audits. Both `lighthouse-cli` and `lighthouse-extension` use this as a back-end. They wrap their input to allow it to be processed by `lighthouse-core`. `lighthouse-core` comes with a `reporter` which generates a report based on the result. This part is used by `lighthouse-viewer` to render the outcome of a Lighthouse evaluation to an HTML report. Most of these modules rely on `lighthouse-logger` to report information to the console. `lighthouse-logger` is a utility module that allows the logging of values in pretty colors and filter messages on priority.

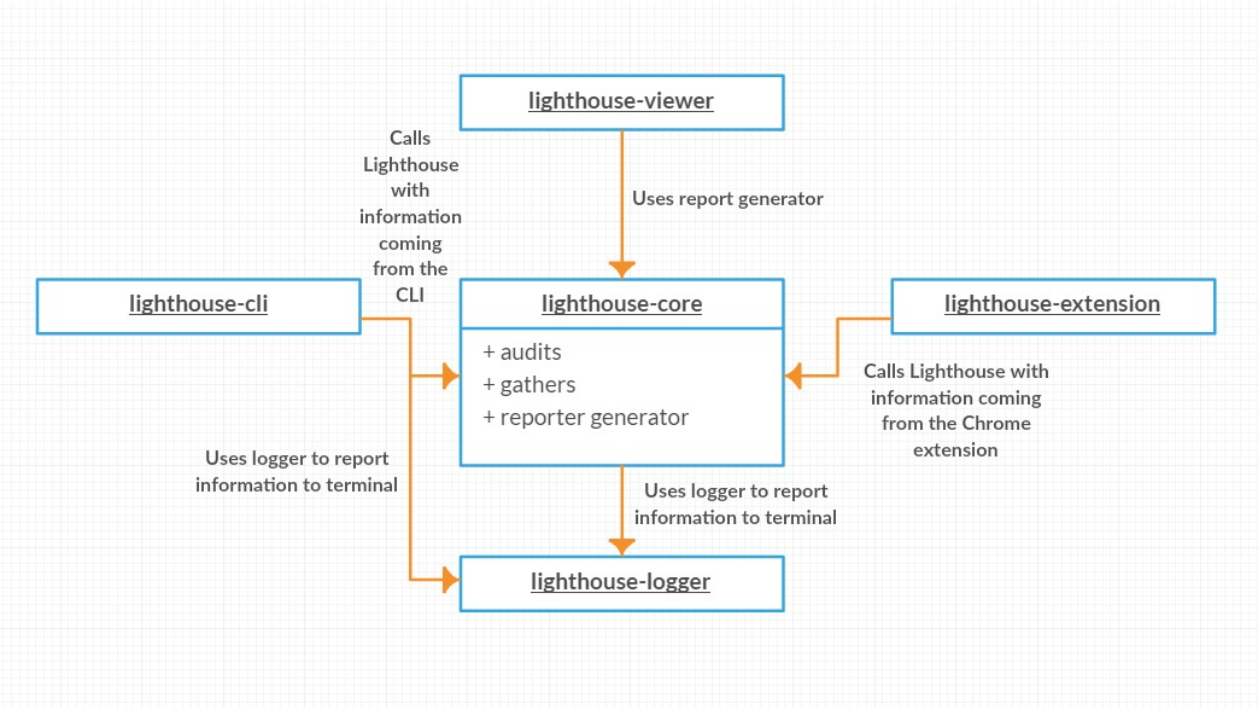


Figure 2: Overview of the module system in Lighthouse.

To further explain the architecture behind Lighthouse we present you Figure 3 which is a chart that is also used by the Lighthouse team [4]. It gives a high level overview of what happens during an evaluation. The 'Lighthouse Runner' is part of `lighthouse-core`. The topics *gatherers* and *audits* will be discussed in section about standardized practices. The website is evaluated in the Google Chrome browser. Lighthouse uses a `driver` which interfaces with the `Chrome Debugging Protocol`. This allows Lighthouse to gather the

required information to perform its tests. Communication between the Chrome instance and Lighthouse happens over a `websocket`. The `driver` sends commands to the browser instance which allows Lighthouse to inject scripts and hooks to control how both the website as well as Chrome behave.

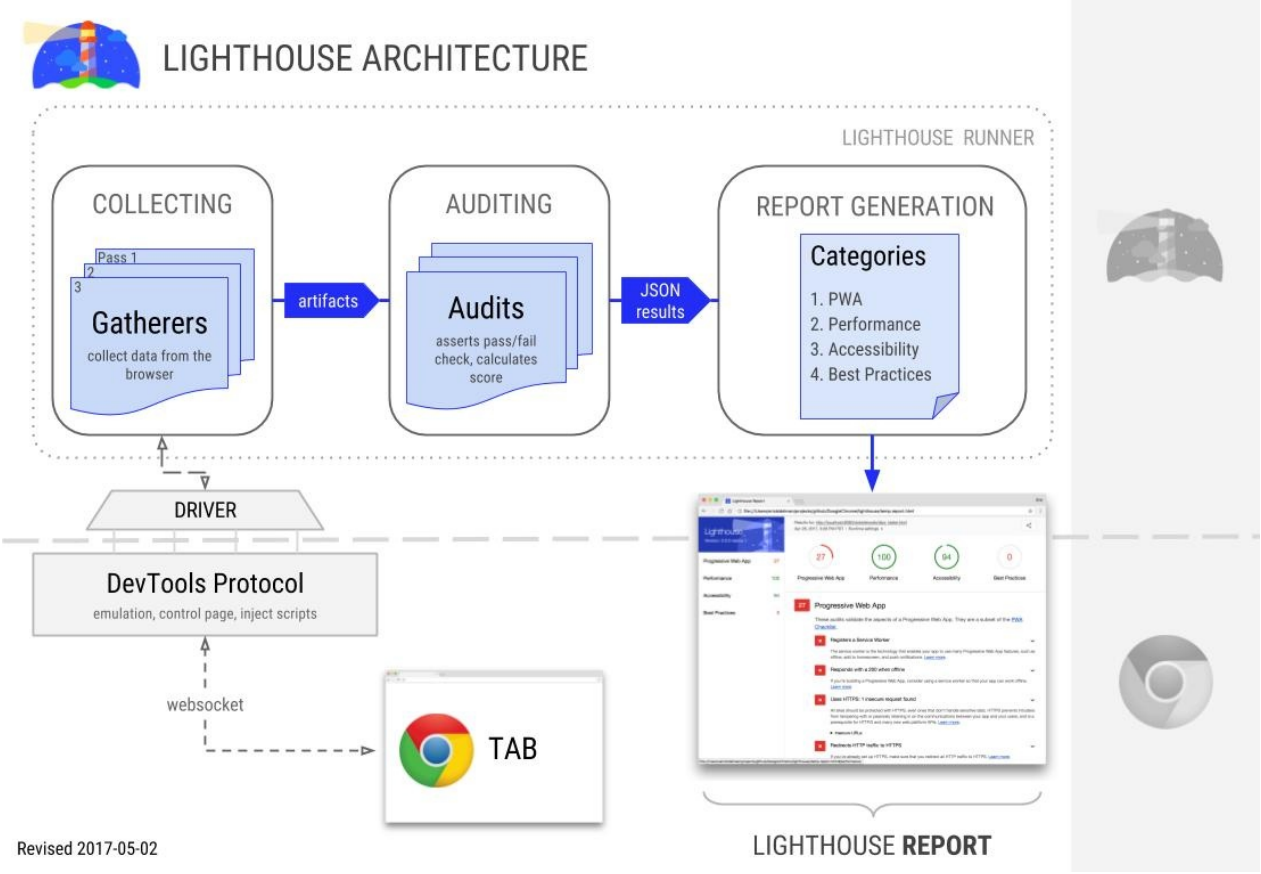


Figure 3. Overview of the architecture of Lighthouse.

Standardized Practices

Because Lighthouse is a project of significant size it is important to standardize certain practices. This in order guarantee that the project remains consistent throughout the development process. In this section we will describe what parts of Lighthouse are standardized and why.

Audits and Gatherers

The goal of Lighthouse is evaluate a website and calculate a performance score. Because of this Lighthouse has numerous *audits* which test one particular aspect of a site. An audit might, for example, be a test which evaluates how long it takes to load the website. But, it is also possible that an audit tests a more abstract concept like if a site is optimized for search engines.

To execute these audits requires some knowledge about a website and how it behaves. In the example of page load it is required to know how long it took the browser to load the page. The required information for an audit is gathered by *gatherers*. These gatherers fetch required information for the audits.

These two interfaces play a major role in the extendability of Lighthouse. Again, Lighthouse is focused on evaluating websites in a variety of ways. However, it might be the case that a developer wants to evaluate a website in a certain way which might not be relevant for other developers. As such, it is possible to create your own audits and gatherers [3]. This gives the developer the ability to create their own tests and evaluate their website on their own criteria. An added benefit is that because of the fact that audits and gatherers are standardized it is fairly easy to import the specialized tests from developers should the need arise.

Standardized Design

Lighthouse is open for contributions from external sources. However, to ensure that the new contributions fall in line with the project there are some rules and standards that apply.

To ensure that all (new) code has a style that is consistent with the existing code base Lighthouse uses a 'linter' to validate the code. The lint rules state what indentation type is used and what sort of statements are allowed. For example, it is not allowed to use `console.log` statements. Instead, developers should use `lighthouse-logger`. These rules keep the code base consistent and clean.

To further streamline the contribution process Lighthouse has a `CONTRIBUTING.md` which states rules that contributors must follow [5]. Lighthouse uses conventional commit for commit messages and pull requests. A bot is then used to assert that all pull requests have a title according to the rules. One important part for new contributors is that they sign the Contributors Licence Agreement (CLA). If new contributors do not sign the CLA their contributions cannot be accepted into Lighthouse because of legal reasons.

Codeline Organization

In this section we will describe how Lighthouse does testing and what its build process looks like.

Testing

Lighthouse testing is highly standardized and automated. As Lighthouse can be integrated with the client's application as a node module, testing becomes a priority to prevent the breakdown of client's application due to unexpected runtime errors in the Lighthouse code.

The testing process consists of two major components:

a) Testing framework:

Lighthouse used modular programming to develop the testing framework i.e each file contains tests related to the specific functionality and in cases where two or more functionalities are integrated, a new file is used. The main tool used for developing the testing framework is `Mocha.js`. `Mocha.js` tests run serially, allowing for flexible and accurate reporting while mapping uncaught exceptions to the correct test cases.

b) Continuous Integration:

Continuous Integration is a software development practice where members of a team integrate their work frequently. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Lighthouse uses `Travis CI`, a well-known service used to build and test software projects hosted on GitHub.

Lighthouse team makes sure that every line of code is covered and validated using automated tests. With every feature introduced to the system, there are test cases added as well, to make sure that all parts of the code are well covered. These pull requests are thoroughly reviewed, in addition to the automated GitHub checks(see Figure 4 Example GitHub checks), in order to prevent the breakdown of the system (See pull request, for example).

Figure 4: Example of GitHub checks.

To make sure the new functionalities integrates well with the older versions, Lighthouse team uses two levels of continuous integration. `Travis CI` runs tests on the commits you push to the repository. Each pull request triggers a minified and development build. And, `Yarn` is used to run all the test modules before releasing/shipping the code. It is done for every release to ensure that the vital parts of

the systems are not corrupted.

```
30  script:
31    - echo $TRAVIS_EVENT_TYPE;
32    - echo $TRAVIS_BRANCH
33    - yarn bundlesize
34    - yarn lint
35    - yarn unit
36    - yarn type-check
37    - yarn closure
38    - yarn smoke
39    - yarn smokehouse
40    # _JAVA_OPTIONS is breaking parsing of compiler output. See #3338.
41    - unset _JAVA_OPTIONS
42    - yarn compile-devtools
```

Figure 5: Travis using Yarn to run tests on each PR.

Building

The Lighthouse team usually releases a new version every month, with occasional minor/patch releases if warranted. A release manager is appointed who is responsible for the release process shown in Figure 6.

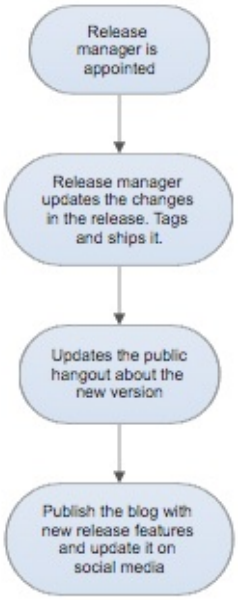


Figure 6: Release procedure used by Lighthouse.

The building process is done using two deployment scripts. The main difference between the two scripts is the visibility of the release. The canary release is done by slowly rolling out the change to a small subset of users before rolling it out to the entire infrastructure and making it available to everybody. Both the scripts use `Yarn` to download all the dependencies and running the test modules on the released version.

Deployment view

As stated in Rozanski and Woods, the Deployment view focuses on aspects of the system that are important after the system has been tested and is ready to go into live operation [2]. It describes the physical environment in which the system is intended to run.

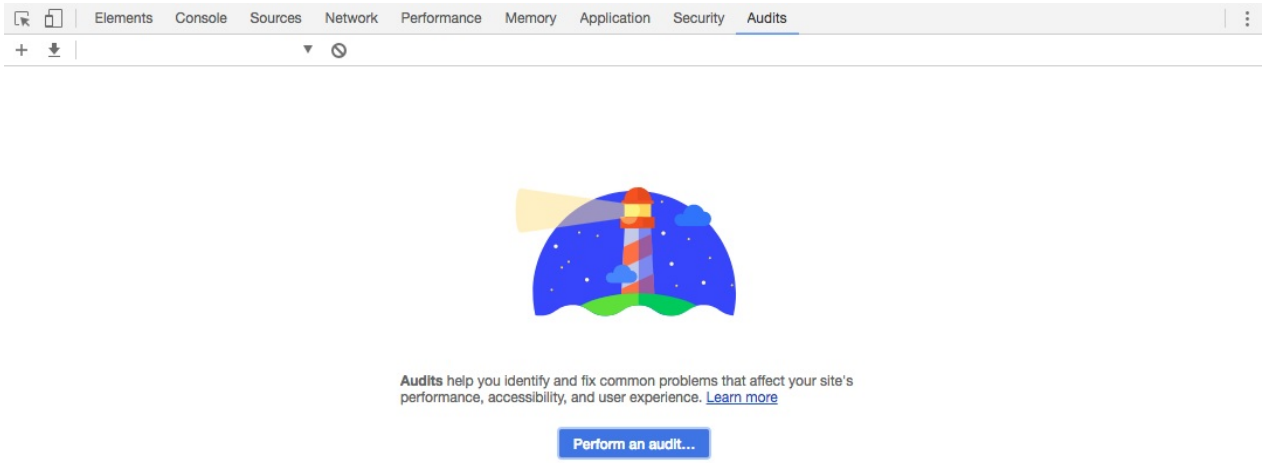
To run Lighthouse the user has three options:

1. within the Chrome DevTools
2. using the Chrome extension
3. using the Node CLI

All three options rely on the usage of (headless) Chrome. In essence Chrome is also part of Google, but since it is maintained by a different division within Google it would still be described as third party software.

Chrome DevTools

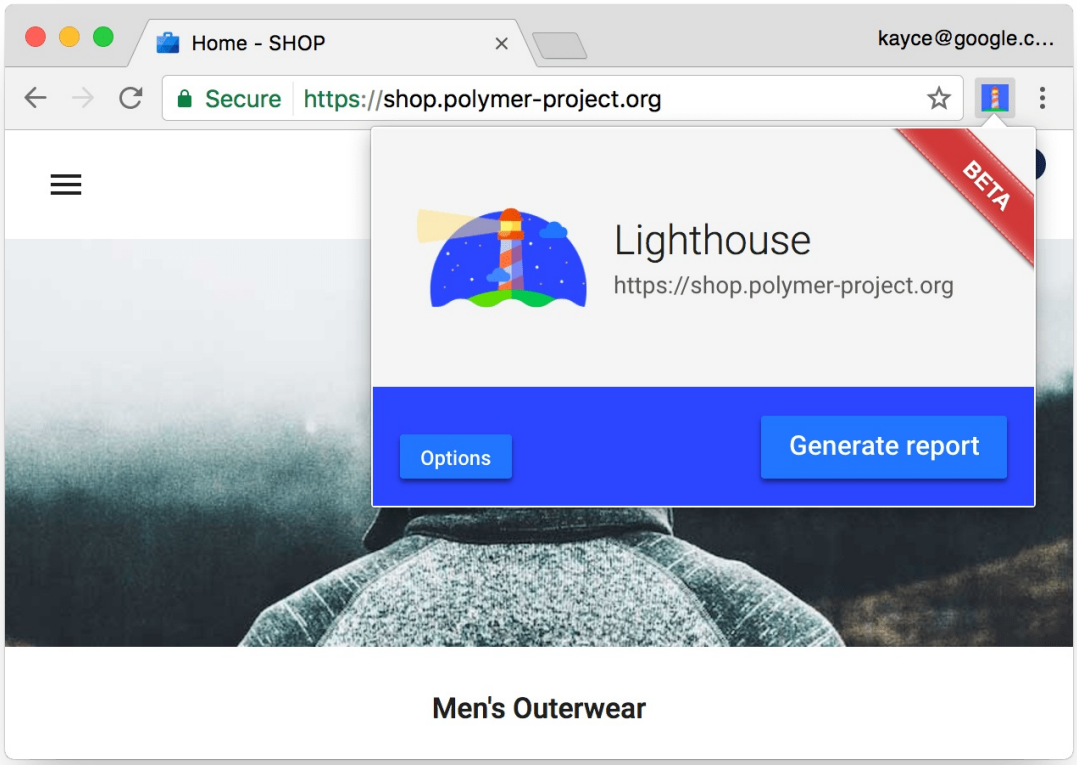
This is possibly the easiest option to run Lighthouse. Since Chrome 60 Lighthouse is accessible from the DevTools. In the audit tab you will find a button to perform a complete scan for the site you are currently visiting. It uses the current website view to perform the analysis.



Using Lighthouse by from Chrome Devtools.

Chrome extension

The Lighthouse team also released a Chrome extension. Installing the extension requires Chrome (obviously). After installation Lighthouse is directly available to perform tests on the current open webview, by clicking on the logo to the right of the URL bar.



Lighthouse Chrome Extension.

Node CLI

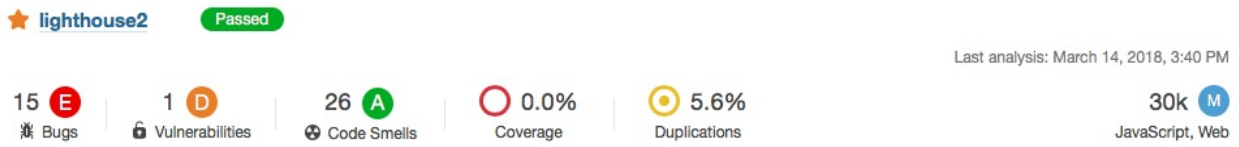
The third option is the most advanced option and requires a little more steps to get it running. The CLI method relies on Node 6 to run, and can be installed by using npm (`npm install -g lighthouse`) or yarn (`yarn global add lighthouse`). Running the tool is easily done from the command line: `lighthouse <url>` . There are many different options to set. One of the more interesting ones is the possibility to use the headless version of Chrome, which helps to use Lighthouse as part of an automated testing chain.

Technical Debt

In this section, we will discuss technical debt and how it affects Lighthouse. We will describe how Lighthouse suffers from both technical debt as well as testing debt. In addition to that, we will address how these two topics have changed over time and how the team communicates about these topics.

Technical Debt

The technical debt analysis has been performed in two different ways: automatically using software tools and by manually inspecting files. The automatic test was performed through the scanning tool from SonarQube. We've performed two checks. The first consisted of the entire Lighthouse repository. This resulted in 300 bugs, 6 vulnerabilities, 351 code smells and 18.6% duplications found in the project. The estimated technical debt by SonarQube was 8 days. However many of the issues were found in less critical parts such as the documentation. So for comparison, an analysis has also been performed on just the lighthouse-core part.



Analysis of Lighthouse using SonarQube.

The technical debt of the core part was estimated by SonarQube to be around 4 hours (3h on major items, 10min on critical items and 25min on minor items). However, the number of hours in itself isn't a good indicator on if the project is well maintained. The ratio between the technical debt and the time it takes for the entire code base to be built gives a better indication. For the Lighthouse-core part, the code base consists of over 30.000 lines of code, and hence the debt ratio, as calculated by SonarQube, is 0%. Which is impressive.

For the manual analysis, we checked the code with for violations of the SOLID principles. The Single responsibility principle is met. All classes are dedicated to specific functions, with a clear naming structure. In the case Open/closed principle we take `lighthouse-core/audits/audit.js` as an example file. It is noticeable that the class can easily be extended with new functionality. All functions within the class are well defined and if required comments are made to clarify. The parameters are described and also the return types are stated. Hence the module is closed. So the file follows the Open/closed principle. This is true for all manually checked files. The Liskov substitution principle can be checked by looking, for example, at the `lighthouse-core/audits/audit.js` file. This file is the parent file of all audits in the folder. Looking at its subclasses, we can conclude that none of them violate Liskov's substitution principle. All subclasses compute a valid override of the parent class. The separation of all parent classes into useful parts also ensures that the code is valid according to the interface segregation principle. No class could be found that requires unnecessary other classes. Looking at the way the abstractions are set up, no violation of the dependency inversion principle could be found.

Testing Debt

Code coverage is a measure used to indicate how much code has been covered by a test. Low coverage implies that the program has a high chance of containing undetected bugs. High coverage does not necessarily signify all actions will be correctly processed by the code, but at least it indicates that the likelihood of correct processing is good [7].

Code coverage can be further divided based on the criteria explained below [8].

1. Function coverage – Has each function (or subroutine) in the program been called?
2. Statement coverage – Has each statement in the program been executed?
3. Branch coverage – Has each branch (also called DD-path) of each control structure (such as in if and case statements) been executed? For example, given an if statement, have both the true and false branches been executed? Another way of saying this is, has every edge in the program been executed?
4. Lines coverage – Has each line in the program were covered by the tests?

Release version	Modules	% Stmts	% Branch	% Funcs	% Lines
2.9.1(latest)	lighthouse-cli	63.5	55.95	50	64.25
	lighthouse-core	97.89	93.18	100	97.78
	lighthouse-extension	100	100	100	100
	lighthouse-viewer	100	100	100	100
2.6.0	lighthouse-cli	33.78	9.88	11.11	33.96
	lighthouse-core	74.55	56.92	84.21	74.53
	lighthouse-extension	100	100	100	100
	lighthouse-viewer	100	100	100	100

Testing at the lighthouse is divided into four major modules i.e cli, core, lighthouse-extension and lighthouse-viewer. The result table presented above clearly reflects the importance given to testing at Lighthouse. All the modules, except lighthouse-cli, cover almost every line of code written for that specific module. On running coverage analysis on prior releases, we found out the testing debt in

`lighthouse-cli` module was introduced during the release of version `2.6.0`. The main culprit file was found to be `sentry-prompt.js`. On further investigation, we found that `sentry-prompt.js` replaced the file `shim-modules.ts`, for which test cases still exists. Given the purpose of `sentry-prompt.js` is to ask user's permission to send runtime error report to Lighthouse, it seems no testing other than manual testing was done for this file.

Apart from the coverage, Testing time is another source of technical debt. The average testing time taken by Lighthouse to run all the tests was found to be around 1 minutes 8 seconds. It seems acceptable as a total of 937 tests were ran during that time covering ~91% of the total code.

Communication about Technical Debt

In this section, we will address how the Lighthouse developers discuss the technical debt. As will be seen, communication about technical debt can be divided up into two parts. First, we will discuss how the Lighthouse team uses Github, and more specifically Github Issues, to discuss the technical debt. After that, we will show how technical debt is denoted in the code base.

Like most discussions about Lighthouse, communication about technical debt happens through the use of Github Issues. Technical debt is never mentioned directly but some of the issues that are raised are a direct result of technical debt. Unfortunately, the Lighthouse team does not use special labels to denote issues related to technical debt. This does make it more difficult to see what caused the technical debt. Searching for pull requests which focus on solving technical debt gives more results. There are a few pull requests where the term technical debt is directly mentioned. However, there are no pull requests which directly mention that they fix the technical debt.

When looking using broader search terms like *refactor* and *maintenance* it is easier to find discussions about (problems caused by) technical debt. Also searching for proposed architecture changes gives good examples of technical debt. What follows is that while technical debt is not mentioned directly the problem is addressed and sometimes solved.

To denote technical debt within the code base the Lighthouse team uses different techniques. Often they use `TODO`s as a way of signifying that a part of the code needs revising. There are 28 `TODO` markers in the whole code base of Lighthouse [9]. However, it is not clear if and when these `TODO`s will be fixed. There is also no indication if there is a matching Github Issue. Therefore it remains to be seen if these `TODO`s will actually be solved.

Evolution of Technical Debt

In this section, we will describe how Lighthouse has dealt with technical debt over time. Technical debt usually accumulates over a period of time. Sometimes it is addressed directly and fixed intentionally. It is also possible for changes to be made to the code base which results in technical debt being paid while not being the primary goal. We will try to give a brief historic overview how Lighthouse has changed over time and how that has impacted its technical debt.

The first version of Lighthouse was released on 30-06-16 [10]. This makes Lighthouse almost two years old. In total 48 releases were made with the most recent version being version 2.9.3. Many changes were made during the two year period and we are going to look at some of them to see how they impacted Lighthouse.

To look at how the Lighthouse project has changed over time we are going to analyze previous versions using SonarQube. This will give an indication how the number of bugs, vulnerabilities and technical debt has changed. Unfortunately, not every version of Lighthouse can be analyzed using SonarQube. We will analyze releases starting from version 2.0 which was released on 30-06-17 [10]. As before, we will only focus on the core part of Lighthouse to avoid including errors in unrelated parts of the code base.

What we found is that in all releases the amount of technical debt was very limited. SonarQube found that the amount of technical debt was only a few hours. This is a relativity low number with respect to the size of the project. But the number of hours of technical debt does not indicate if the project is being maintained correctly. As has been mentioned before, The ratio between the technical debt and the time it takes for the entire code base to be built gives a better indication. With all releases, this ratio was 0%.

It follows that the amount of technical debt in Lighthouse is fairly limited. While there are definitely instances where decisions from the past have a negative impact on future development the actual extend of these decisions is limited. The project is not being held back by a huge amount of technical debt. New additions to the project can happen organically and the code base changes over time to meet the needs of the developers.

Evolution perspective

One of the great benefits of software is its ability to evolve very quickly. For Lighthouse this is very relevant, since web techniques are highly subjected to change. Since the first commit by Paul Lewis to the Lighthouse project in January 2016, Lighthouse is now currently at version 2.9.3.

Timescale and Likelihood of Change

Thanks to the rise in popularity of the Progressive WebApps (PWA) the web landscape is changing. Web users expect a native feeling to web applications, as well as full functionality. Features that used to be dedicated to native smartphone apps (e.g. notifications, offline capabilities, hardware acceleration, etc.) are now moving to the web. New Javascript and CSS capabilities are developed to keep up with this trend, resulting in 3 different Chrome versions launched in March 2018 [11]. Being an automated web checking tool, Lighthouse needs to evolve continuously to stay relevant. However not all changes are just related to the evolving landscape of the web, new features are continuously added to make the tool more convenient for the users. The focus for changes resolves around the four main audit categories:

1. PWA Checklist
2. Best Practices
3. Performance
4. Accessibility

It is safe to say that the timescale for changes is rather short, and the likelihood of changes is high.

v2.0.0 and Operation Yaquina Bay

One and a half year after the first commit, Lighthouse v2.0.0 was released. A lot of work had been done. The first most visible one was the User Interface change. Making the results more clear by adding the now characteristic gauges, screenshots, filmstrips, sparklines, accessibility by sections and, pass and failures separation. Version 2.0.0 also contained the integration with the Chrome DevTools, making the tool easier accessible to many web developers.

Another important part of the 2nd version of Lighthouse is operation Yaquina Bay. The operation is named after the Yaquina Bay lighthouse which, like a fast webpage that you can see it load quickly from start to finish, is short enough that it doesn't take long to see the whole thing.

Yaquina Bay.

The goal was to speed up Lighthouse, or as Paul Irish put it: "I mean.. a performance tool should probably be performant. ☺". Nine issues were fixed, resulting in a good performance boost: a full run for cnn.com went from 169s to 50s (238% faster), and theverge.com went from 92s to 53s (73% faster).

The rest of the v2 update consisted of many fixed issues and added features. A short summary: Audits (14 commits), Metrics & Precision (17 commits), Plots (4 commits), CLI (3 commits), Testing (6 commits), Misc (12 commits), Docs (9 commits) and Deps (4 commits).

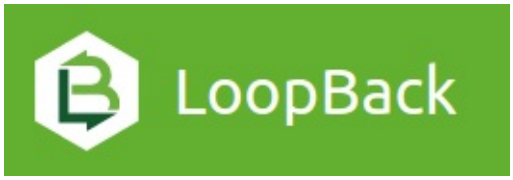
Future evolution

Since the v2.0.0 release in August 2017 many more releases have been made bringing the current version to 2.9.3. In the issue tracker the 3.0 label is visible, and used by the team, so a new big release can be expected.

References

1. The System Context Architectural Viewpoint, Eoin Woods and Nick Rozanski, http://www.artechra.com/media/writing/WICSA2009-context-view-paper.pdf
2. Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012.
3. Creating custom audits, https://github.com/GoogleChrome/lighthouse/tree/master/docs/recipes/custom-audit
4. Lighthouse architecture description, https://github.com/GoogleChrome/lighthouse/blob/master/docs/architecture.md
5. Lighthouse rules for contributors, https://github.com/GoogleChrome/lighthouse/blob/master/CONTRIBUTING.md
6. Lighthouse release process, https://github.com/GoogleChrome/lighthouse/blob/master/docs/releasing.md
7. Brader, Larry; Hilliker, Howie; Wills, Alan (March 2, 2013). "Chapter 2 Unit Testing: Testing the Inside". Testing for Continuous Delivery with Visual Studio 2012
8. Code coverage on Wikipedia, https://en.wikipedia.org/wiki/Code_coverage
9. TODOs in Lighthouse, https://github.com/GoogleChrome/lighthouse/search?utf8=%E2%9C%93&q=TODO&type=
10. Lighthouse releases, https://github.com/GoogleChrome/lighthouse/releases?after=1.1.2)
11. Google Chrome Version History, https://en.wikipedia.org/wiki/Google_Chrome_version_history

LoopBack



Project by Yann Rosema, Claudio Lazo, Wesley Quispel and Joost Wooning.

Delft University of Technology

Abstract

LoopBack is a framework which allows for quickly creating APIs on existing databases. A new version of LoopBack is expected soon, the development started in January of 2017 from scratch. Since than the LoopBack team has tried to keep a clean and understandable codebase. In this chapter we will analyze the architecture of the LoopBack project. This also includes analyzing older versions of LoopBack to be able to understand why the LoopBack team has chosen to rewrite the entire project.

Table Of Contents

Introduction

StrongLoop's LoopBack is an open-source Node.js framework that allows developers to easily build connective applications. Through the command line interface, within minutes a skeleton for the application is created. The program generates a REST API, has built in role-based access controls and automatically creates models (classes) and relations based on CLI input and it connects easily to data sources.

LoopBack is currently in the development state for version 4, or LoopBack4, the first release is expected June 2018. LoopBack4 is rewritten from scratch therefore it gives us the perfect opportunity to find out why the StrongLoop team decided for this rewrite. We will try to get a good insight in the architecture of LoopBack, for this we use the methods described in Rozanski & Woods (2012). Since the codebase is relatively new (the development started in January of 2017) we expect that the project architecture is quite good.

Stakeholder Analysis

To be able to understand the role of the stakeholders architecture in the development of LoopBack is fundamental. In order to have insights in the stakeholders and their roles, a stakeholder analysis is required. This is done according to the 11 stakeholder types of Rozanski & Woods (2012).

Stakeholder types

Rozanski & Woods identify eleven stakeholder types and for LoopBack we also identify three other stakeholder types. An overview of these stakeholders is shown in table 1.

Type	Stakeholder	Description
Acquirers	IBM, StrongLoop management, founding developers (such as Raymond Feng)	The procurement of LoopBack was overseen by the developers who perceived a utility in its existence.
Assessors	IBM's legal department, System architects	IBM is responsible for legal regulations and the system architects for adherence to software standards.
Communicators	Developers, maintainers via GitHub and Jekyll	The people that maintain the documentation and the website are the main communicators. This is most often done by the maintainers.
Developers	Raymond Feng, Miroslav Bajtos and Ritchie Martori	The construction of the system from specification is done by a highly active team followed by less active contributors, further deployment is completed by the users of LoopBack.
Maintainers	Biniam Admikew, Diana Lau, Janny Hou, Simon Ho, Kyu Shim, Taranveer Virk, Yappa Hage and open-source contributors	Further developers that maintain LoopBack, create the documentation and debug it.
Production Engineers	-	As this software project has no production phase but only a development phase, there are no production engineers involved.
Suppliers	OS (Windows, MacOS, Linux), Github, Node.js, NPM amongst many other dependencies	LoopBack has many dependencies, which can all be considered suppliers.
Support Staff	Developers, contributors, IBM Support Portal	Support can be formal in the form of IBM's support or contacting the development team or more informal using third-party sources like StackOverflow.
System Administrators	Users	Since the users will be responsible for deployment, system administration will fall to them.
Testers	Developers, Users	Issues that are discovered can be disclosed on the repository. This means both users and developers are involved in testing. LoopBack also uses continuous integration.
Users	API builders for GoDaddy.com, Symantec, Sapient, et al. Also the end-users that use applications built using LoopBack	The users that define the system's functionality and ultimately make use of it.
End users	People who use an API created with LoopBack	These people want a consistent use of the API, which is conform the REST specification.
Standardication organizations	OpenAPI	OpenAPI is an industry standard for the design of REST APIs. loopback-next integrates the OpenAPI standard using multiple packages to make sure the rendered API complies with this standard.
Tool providers	Visual Studio Code	The loopback-next team plans on adding new tools that work with Visual Studio Code planned_features making this software a stakeholder of the project.

Table 1. LoopBack stakeholders sorted by type

Power Interest Grid

The power-interest grid in figure 1 maps certain stakeholders on a 2D-plane according to the power they have over the project and their interest in the project. In the top right corner we see the developers of LoopBack who have the most power and the most interest in the success of the project.

With a little less power and less interest, we have the maintainers who work on the software but need reviewing by the developers. With again less power we have the contributors who write code which needs to be reviewed by the developers or the maintainers. IBM and its legal department have a similar level of power but less interest because LoopBack is software from IBM but as it is a big company, they have less interest in this project. The suppliers have (close to) no interest in the project as they do not play an active role in it, however they do allow the project to keep on living which means they have a non-zero amount of power over the project. The users have a high interest in the software and can, as a community, ask for changes, but as long as they do not become contributors, their power remains low. The IBM support portal has a similar amount of power to the user, because they can ask for changes in the software. However because they are part of IBM, LoopBack is not their main focus. Lastly, the competitors have a high interest in the software, but almost no power.

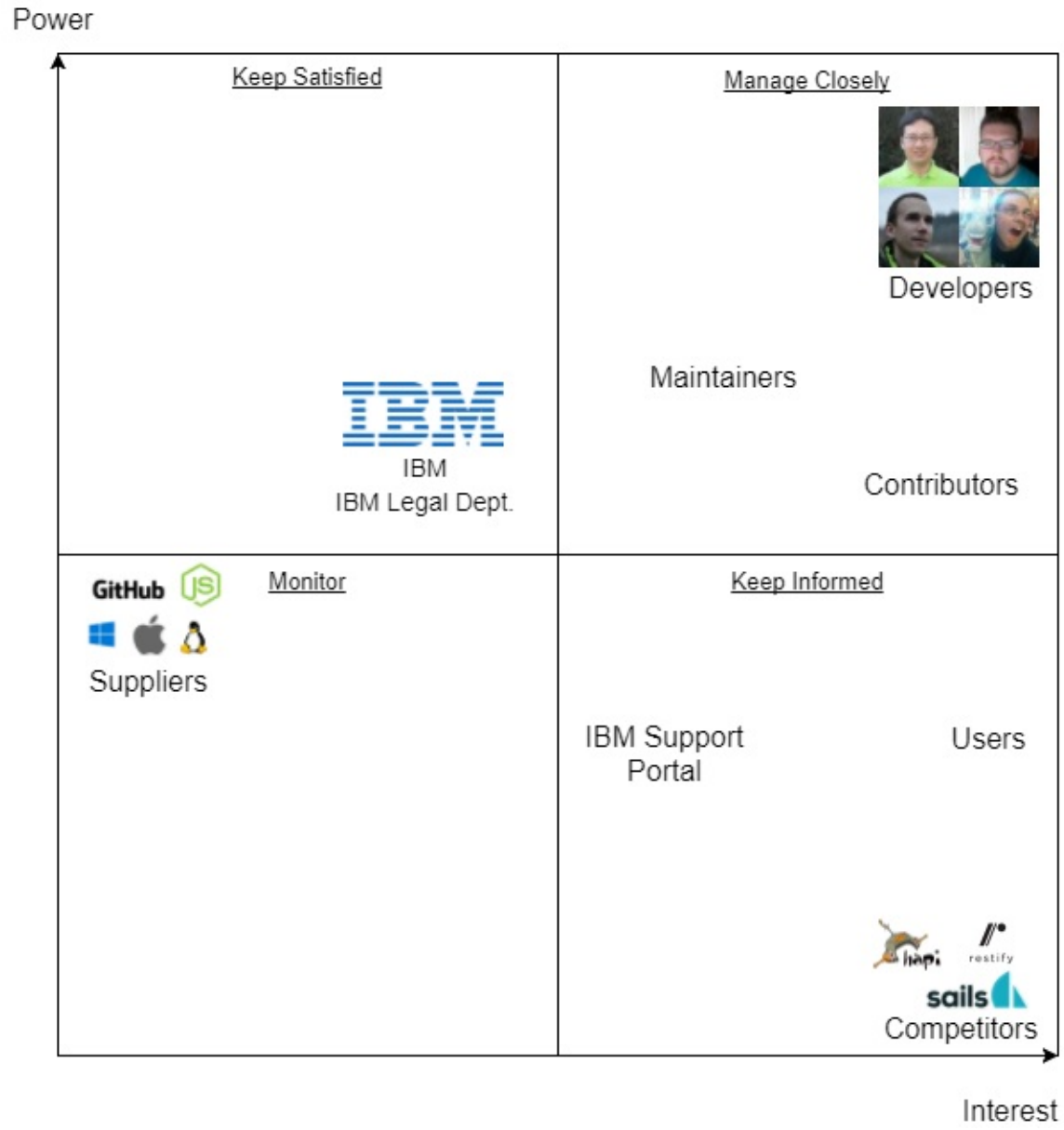


Figure 1. Power interest grid of the LoopBack software.

Context View

This section describes all external entities (systems, people and organizations) that LoopBack interacts with. The goal of this analysis is to identify dependencies and important relationships with its environment. The entities are shown in the context view of figure 2.

Scope & Responsibilities

LoopBack is a JavaScript REST API framework; its scope includes, but is not limited to, the following functions:

- LoopBack generates a REST API skeleton
- LoopBack allows for the customization of the REST API through a command line interface
- LoopBack has an API explorer (UI)
- LoopBack supports multiple JavaScript packages
- LoopBack provides role based access control
- LoopBack is compatible with multiple databases
- LoopBack allows HTTP request routing
- LoopBack generates standard models

In order to narrow the scope of the software, the following functions specify what the software does do:

- LoopBack does not include a GUI
- LoopBack does not provide frontend services (other than the included API explorer)
- LoopBack does not limit the rate of accessibility

External entities

Besides the identified stakeholders, many other entities surround LoopBack and can be of influence. They are divided into three categories: market, development and technology.

1) Market

- *Owner*: StrongLoop, an IBM-owned company, is the company that created LoopBack. See the stakeholder 'acquirers' for more information.
- *Competition*: There are many other frameworks for the creation of REST/SOAP APIs, like Meteor, Restify, Dreamfactory, Sails, Hapi and many more. This means that LoopBack must keep innovating in order to stay in this market.
- *Users*: The users can be any person or organization, for example GoDaddy, Symantec and Bank of America.

2) Development

- *Support*: There are many open source support channels like Gitter chat, GitHub, Google Groups and the LoopBack FAQ & Documentation. For enterprise clients (e.g. using API Connect) there's a professional support department.
- *Developers*: The lead developers (Raymond Feng, et al.) are in charge of the project. They decide to merge pull requests, what will be worked on next and what stays in the backlog. The GitHub community helps with maintenance and development.
- *Development area*: The project uses Git as a versioning system and GitHub for project management and as a Git controller.

3) Technology

- *Platform*: LoopBack is written in TypeScript and runs in Node.js. It is developed in accordance to OpenAPI, a standard for APIs.
- *Software dependency*: It is dependent of the Node.js package manager NPM for installing all dependencies, as the default LoopBack REST API already uses 500+ packages. As an API connects data sources to the web, LoopBack is also dependent of the software from the data source (MySQL, Postgres, MongoDB, Cassandra, etc.). NPM installs connectors for these data sources.
- *User validation*: LoopBack has a default access control functionality which can be expanded with OAuth2 for using a Google/Facebook/GitHub account to access the API.
- *Distribution*: A LoopBack server can be deployed on Windows, Linux, OS X, Solaris and in Docker
- *License*: LoopBack has been given an MIT license for Open Source.

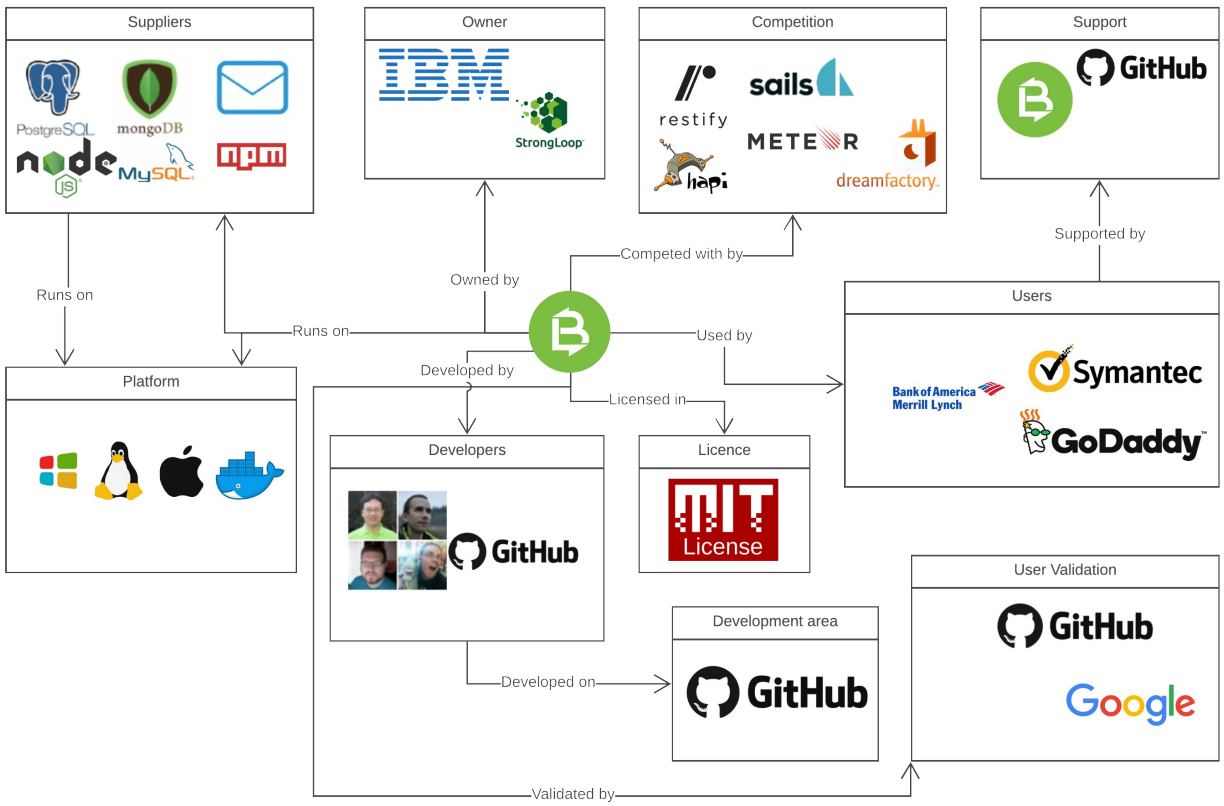


Figure 2. The context view for LoopBack

Development View

The goal of the development view is to describe the architecture that supports the software development process (Rozanski & Woods, 2012). Large-scale software projects benefit from ordered codespace and standardized processes, providing maintainability, reliability and technical cohesion. The development view consists of 3 parts:

1. Module structure
2. Common design
3. Source code structure

Module structure

The loopback-next repository is divided into packages. These packages represent the different modules present in the repository. The packages are managed by [Lerna][lerna] and NPM scripts are used to work with Lerna.

The list of packages from [loopback-next/MONOREPO.md][monorepo] is used as the modules of the system.
This list can be separated in different groups of modules. We identified 4 groups:

- LoopBack4: these modules are used to build and test the code
- API development: these modules are used to create the API for which LoopBack is the framework
- API resources: these modules are used as resources and resource management for the build of the API
- UI: these modules are used for the user interface

How the modules are classified in these groups can be seen in figure 3. This figure also show the dependencies of the modules based on the NPM dependencies in `package.json` located in every package.

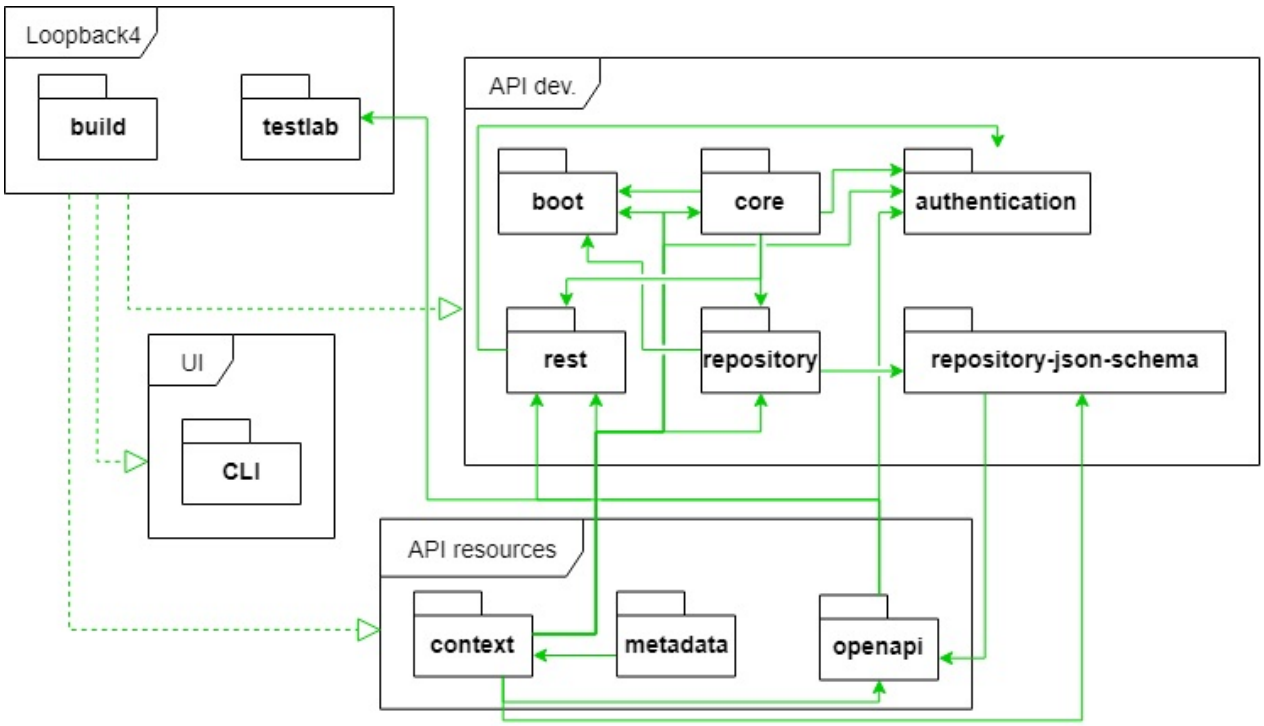


Figure 3. Code module dependency (all openapi packages are categorized under 'openapi').

The build package creates the LoopBack software. It can be run by the testlab package which will build the code and test it afterwards. Once the software has been built, the user can run it by calling it from the command line. The user commands refer to code contained in the API development packages which, in its turn, refers to the API resources to customize the users product.

Common design model

LoopBack4 is being designed with a YAGNI (*You Ain't Gonna Need It*) mentality, which entails that you design and build for what is needed at present and not in the future. Since there is a diverse group of people with different perspectives on API creation, focusing on the MVP (*Minimal Viable Product*) allows the team to address the root issues and build the absolutely necessary components first, before creating other requested features.

To analyze how LoopBack4 tries to achieve this developmental approach we use a common design model. This model consists of three parts according to the Rozanski & Woods, namely:

- *Common processing*, i.e. identifying tasks that benefit greatly from using a standard approach across all system elements.
- *Standard design approaches*, i.e. identifying how to deal with situations where implementations of a certain aspect of an element will have a system-wide impact.
- *Common software*, i.e. identifying the software that is used in a way that it can reduce development time and risk, and explaining why.

Standard design approaches

Coding standardization

Because LoopBack4 is developed by a large group of contributors, but a small group of developers decides which code makes it into the master branch, it is necessary to have a coding style to maintain code standardization and legibility.

This style guide is provided on the LoopBack website. Examples of guidelines are the *prescription of variable declaration* (use of constant variables), *the use of arrow functions*, *using syntax-sugar for class declaration*, *using one argument per line*, and *single line indentation of multiple-line expressions in return, if statements, or multiline arrays*.

In addition, a wiki discussion took place regarding the case to be used in the code. The conclusion was to use kebab-case in names and dots to separate filenames from the file type (e.g. index.d.ts). This decision was discussed in pull request #290 and should resolve problems with different case sensitive operating systems.

Software patterns

LoopBack4 is built using the Model-View-Controller pattern, "where code responsible for data access and manipulation is separated from the code responsible for implementing the REST API".

An example for using this pattern is given, where the facade is the top-level service that serves the *account summary* API, which depends on three services *Account*, *Customer*, and *Transaction*. The facade only aggregates these three services and is independent from their functionality. Thus, it is possible to define APIs in the way that is required by the deployer; Data access and manipulation code is separated from code responsible for client side APIs. To create this, decorators or controllers are used.

Source code structure

It is important to analyze the software on a low level (the source code and development processes), as mistakes on this level can muddy the interactions/dependencies between modules and commonalities of the design.

LoopBack is a complex piece of software, containing many dependencies. There is a lot of code, both JavaScript and TypeScript, and the source code doesn't even contain the required npm packages. This makes it important to structure the source code in an intuitive way.

Analysis

The source code of LoopBack is very organized. Clearly the developers designed this top down by creating modules (see section Module structure) and sticking to it during development. As a result, the *packages* folder gives a good overview of the separate modules and their responsibilities. Figure 4 gives an overview of the source code.

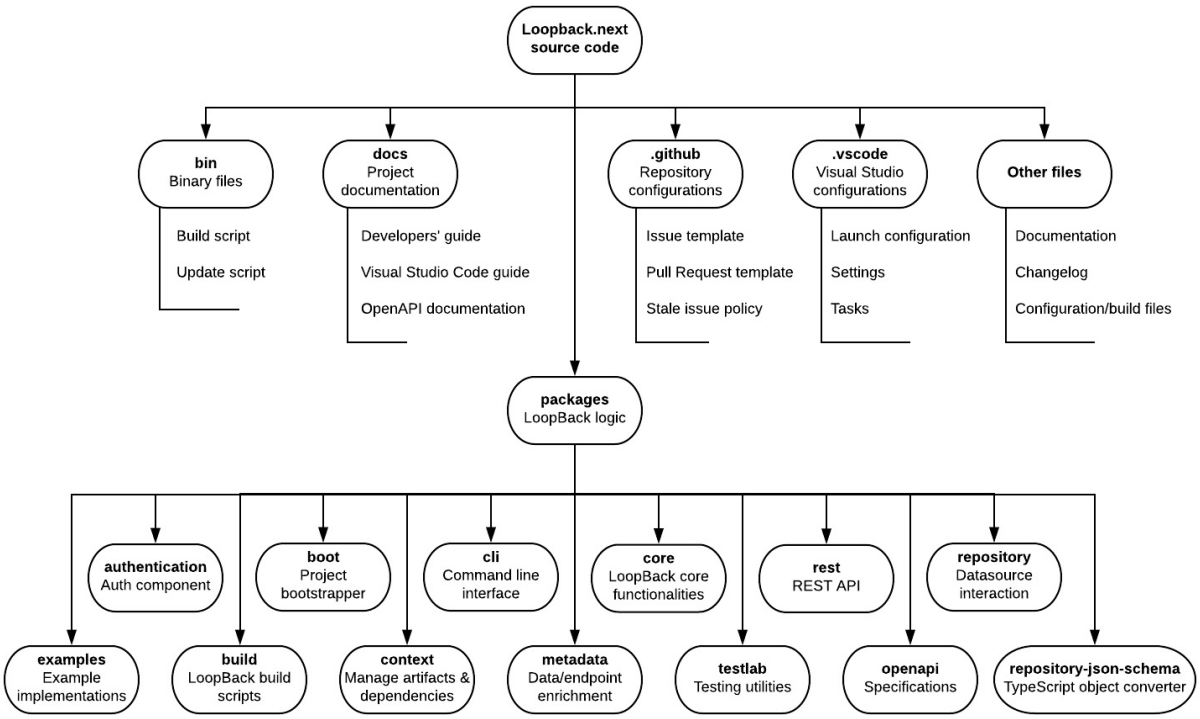


Figure 4. An overview of the source code structure.

The *packages* folder contains the LoopBack logic. Most packages have the same structure: a source and test folder, an extensive `README.md` file and `index.ts/index.js` files. Using this template makes it easier for people to understand the commonalities of these packages.

Discussion

By having tests per package instead of one giant test folder, the source code is much more modular, flexible and easier to maintain. However, this gives the directory a higher tree depth, reducing understandability.

Issue #836 brought the examples into the monorepo (*/packages/examples*). However, the *packages* folder is already substantially sized. We suggest to make a dedicated examples folder in the main project folder, as these have nothing to do with the actual LoopBack REST API logic in the *packages* folder, but just show how to get started or work with LoopBack4. (This has been proposed as issue #1218 , the developers acknowledged it and we have made PR #1231).

Development process

Since LoopBack4 is still in a development state (major version 0) there are many changes which break backwards compatibility. The development of LoopBack4 started in January 2017, the target for the first release is June 2018.

Building

Because of the monorepo structure LoopBack uses, it is preferred to be able to test and build all packages at once. For this exact purpose LoopBack uses [Lerna][lerna] this is a tool to optimize workflow in JavaScript projects with multiple packages. When building all the individual packages many first need to compile TypeScript files to JavaScript. After this the building script detects which EcmaScript version it has to build to and transpiles this to the correct version if needed. Finally the NPM builder actually builds the packages.

Testing

To ensure the quality of code before committing LoopBack allows for testing using the Mocha testing framework and while it is not required to do test driven development, for all pull requests it is required to also add tests where necessary. To measure code coverage [Coveralls][coveralls] is used, this also allows for a overview of how the coverage changes over time.

Guidelines that prescribe standard testing guidelines are mentioned in the testing style guide. Some of the guidelines mentioned in this guide are:

- Using sandbox directories
- Email examples should use the `email@example.com` format
- Correct usage of hooks with for example `before` and `after` statements
- The layout of test files
- Callback function next line prescription
- Naming of files and functions

To test the project the `npm test` command is called which will build and then test the implemented functions.

Use of GitHub

StrongLoop uses GitHub as the main communication channel. For a structured use of GitHub they use the conventions listed below.

Milestones

StrongLoop uses the milestones for keeping record of a sprint backlog. They used to work with sprints of a week, but recently changes to monthly sprints.

Pull Requests

Pull requests are used to review code contributions, this happens with the following conventions. First of all, for all pull requests there is at least one of the core developers which reviewed the pull request. Secondly, for a structured check for pull requests they have provided a checklist which should be passed for each pull request. The checklist consists of (copied from GitHub):

- `npm test` passes on your machine
- New tests added or existing tests modified to cover all changes
- Code conforms with the style guide
- Related API documentation was updated
- Affected artifact templates in `packages/cli` were updated

- Affected example projects in `packages/example-*` were updated

Furthermore there are continuous integration tools implemented which run on every pull request automatically they cover the tests, the code linter and more, some of these are required to merge the pull request.

Releases

Because of the monorepo structure, releases are separated per package. However, all packages are released at the same time. It seems from the release timeline that there is no defined interval in which everything is released but it is approximately twice a week. Since LoopBack4 has no real full release yet we can only look at LoopBack3 for this, but it seems they use the same kind of release cycle once the package is no longer in a development state.

Commit messages

For commit messages this project uses [Conventional Commits][conv_commits] in combination with commitlint to enforce these conventions. To simplify the use of conventional commits the use of [commitizen][commitizen] is encouraged. The point of this style of commit messages is that it is easy to work through the project history and that changelogs can be automated.

Technical debt

Technical debt is a metaphor for an outstanding debt due to the implementation of technical solutions that are not maintainable or evolvable. The metaphorical debt consists of costs that have to be made to refactor the code in order to realize a consistent and maintainable solution (Cunningham, 1992).

loopback-next includes a linter, namely TSlint. This is a code analyzer that looks at bugs, formatting and programming errors. TSlint is highly customizable so the developers added their own rules to make sure the code follows their make up and utilization rules.
For the completeness of the analysis, another tool is used to estimate the technical debt. The analysis done by SonarQube reveals what was already expected: the code has very little technical debt.

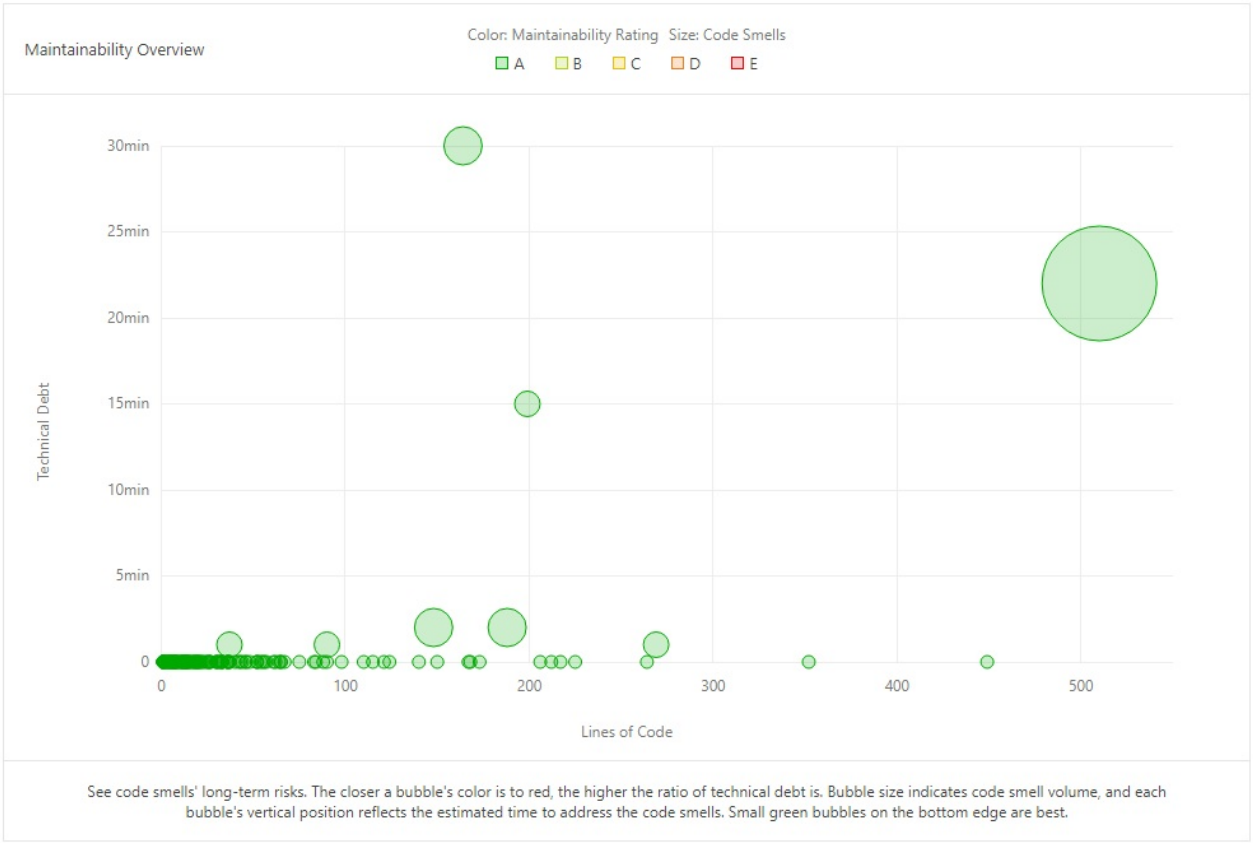


Figure 5. Technical debt analysis by SonarQube.

Figure 5 shows the result of the SonarQube analysis. Each circle shows a code smell picked up by the analyzer. As can be seen all dots are green meaning the code smell's impact on the overall code is very small. After analyzing 7.7k lines of code, SonarQube only found 18 code smells with a total of 1h14min of technical debt (SonarQube measures the amount of technical debt in the estimated time it takes to fix it). The technical debt mostly consists of unnecessary cast and some functions with a different name but the exact same content.

While trying to write a fix for a tech debt issue involving moving some packages out of the monorepo, we came across some potential technical debt. Here we find a piece of code download the whole loopback-next repository from github again in order to have an up to date version of the examples. This code feels messy and not finished, as the comments in the file mention assumptions on folder and file names which - if they fail - break the code's functionality. Also, in order to access an example package, the code downloads all packages which takes a long time and is not needed. This piece of tech debt was found in the JavaScript part of the loopback-next code which in general does not appear to be as well written and commented as the TypeScript part of the code.

Finally, the amount of technical in the project can best be analyzed with the help of the issues on GitHub. In the repository, the team has included a label `tech-debt` with which they label issue they consider being a technical debt. The team also uses a label named `refactor`. Under this label, the developers class issues that involve refactoring but are not considered to be technical debt. This is, for example, the renaming of file to match a company wide format or remove node modules that are not used anymore and are thus dead code.

Impact

Identified technical debt is: unnecessary casts, functions with a different name but the exact same content and some small code smells. The impact of this identified technical debt is reduced future extensibility, whereas the developers want to create a highly extensible application. However, there is so little technical debt due to the redesigning of LoopBack with technical debt in mind, that the impact is low.

As mentioned on their website, one of the reasons for rewriting LoopBack4 from scratch, is the accumulation of technical debt in LoopBack3. This manifested itself mostly in the form of opposition to scalability. LoopBack3 was becoming more and more diverse and needed to account for a lot of variability which resulted in backwards compatibility issues and multiple pieces of code with the same action making it more complex to handle.

Testing debt

In this section, technical debt due to inadequate testing is identified. For this we will look at the testing coverage provided by [Istanbul][istanbul]. We will also make some recommendations according to our findings.

We did a sample test to check the quality of the tests and it seems that every test does indeed test something new. However, we found some tests were it has some TODOs in the testing code, thus the tests could still be improved.

Testing coverage

For testing the coverage LoopBack uses Istanbul, this works in combination with the [Mocha][mocha] testing framework. Unfortunately the versions for both Istanbul and Mocha differ for LoopBack3 and LoopBack4, but we assume the results can be compared throughout these versions. We compared the testing coverage for both LoopBack3 and LoopBack4, see table 2 for the results. It is clear that the coverage for LoopBack4 is better then for LoopBack3, however this makes sense knowing that LoopBack4 has a much shorter lifespan.

Coverage	**LoopBack3**	**LoopBack4**
Statements	88.4%	97.3%
Branches	80.3%	89.7%
Functions	88.0%	95.0%
Lines	90.1%	97.4%

Table 2. Test coverage from both active versions of LoopBack.

All the individual packages for LoopBack4 also contain tests which can be individually tested with `npm test`. These tests for the packages are divided into three different kind of tests: unit, acceptance and integration. However not all packages contain all kind of tests, this might also be a good improvement for this project.

Testing Packages

Since LoopBack4 has a clear distinction between the different packages we can also compare the results within LoopBack4.

For example, the test coverage of the small-sized `example-rpc-server` package is only around ~62 percent, while the similarly sized `repository-json-schema` package is fully covered by testing. The most crucial `core` package is covered for 97 percent, with 99% of the statements, 95% of the branches, 96.8% of the functions and 98.9% of the lines covered by testing. Overall Coveralls reports a test coverage of around 96% for the entire codebase.

There are two packages which are different from the others when it comes to testing, these are the cli and build package. The cli package does not show up in the coverage results by default, the tests do however run. What can be seen in table 2 for the cli package is what we found out ourself, the coverage is lower then other packages, but still pretty high. We think it would be an improvement to include the cli package in the testing coverage to keep track of the coverage for the package.

The other different package is the build package, the tests for this package are about 20 integration tests. However, these integration tests are not run by `npm test`, therefor these are not tested as often as the normal tests. We think these tests should be included in the normal testing procedure, since we could not think of a reason not to.

Coverage timeline

For coverage reporting LoopBack uses [Coveralls][coveralls], on the Coveralls website you can find the report for loopback-next. In the report for the loopback-next repository you can see a bump about half a year ago, from the commit message for that particular bump you can see that the testing framework dependencies were updated. A dip somewhat further shows a commit where the support for nodejs6 was dropped. It seems that for LoopBack4 the coverage has always been quite good, thus there is no real need for improving the coverage by adding more tests.

Evolution of technical debt

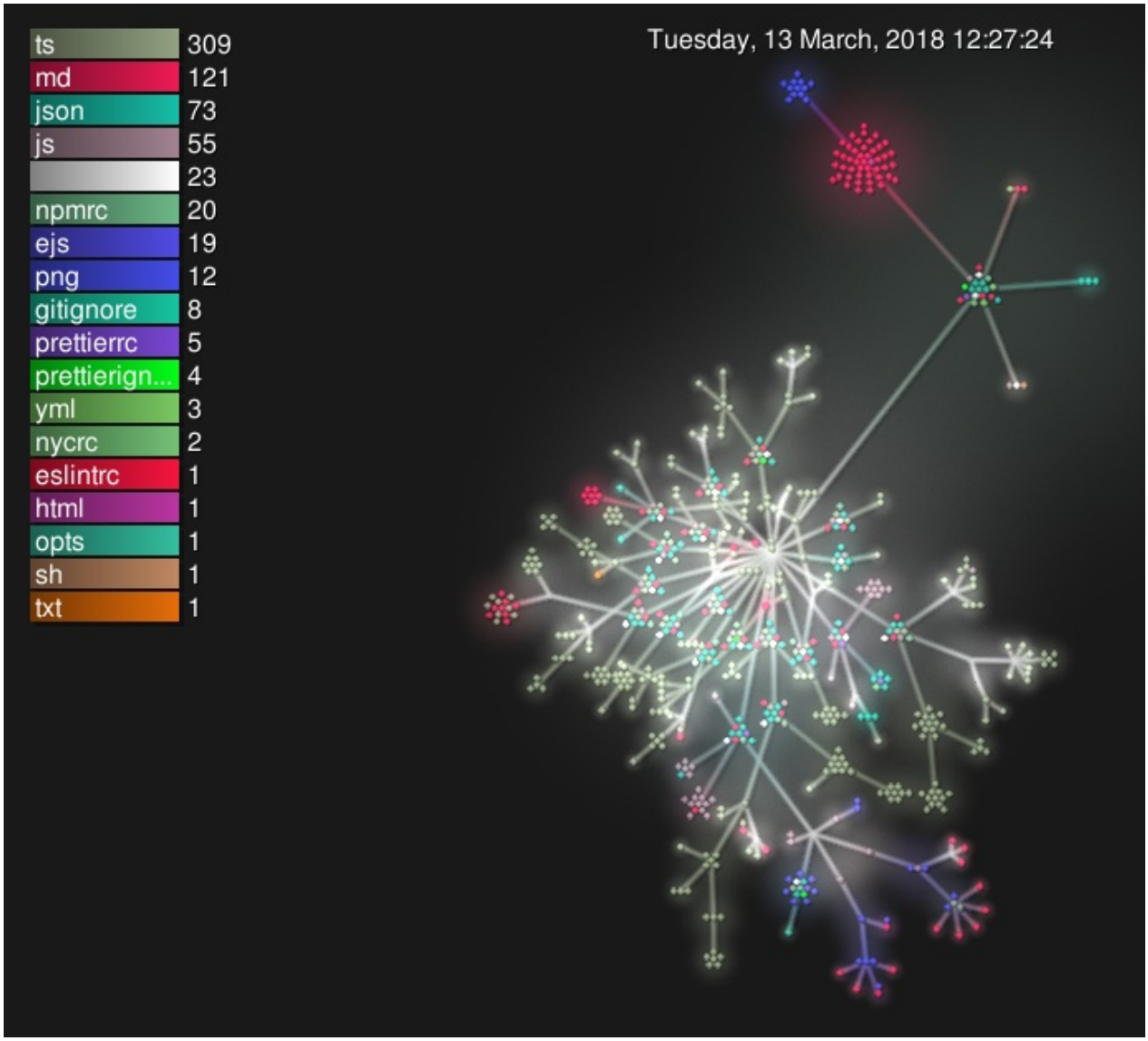


Figure 6. Overview of the `loopback-next` repository at the latest commit.

Scope Definition

LoopBack4 has only been in development relatively shortly as a rebuild of the previous LoopBack3, this is since 9 January 2017. As such, there have not been as many commits or files as other more mature projects. The files that are committed the most are the package files which maintains the version numbers for dependencies in building of the application. However, since these files have no importance on the code quality these are omitted from this analysis. Instead, a short overview of the 20 most committed files in `loopback-next` can be seen along with the amount of active days these files have been changed. This table was obtained by using the `git effort` command.

Class	Amount of Commits	Active Days
packages/core/src/application.ts	37	33
packages/context/src/inject.ts	33	22
packages/context/src/resolver.ts	33	22
packages/core/src/index.ts	30	25
packages/context/src/binding.ts	30	22
packages/context/src/context.ts	28	22
packages/context/test/unit/resolver.test.ts	24	16
packages/context/src/index.ts	24	20
packages/context/test/unit/context.ts	23	20
packages/authentication/test/acceptance/basic-auth.ts	23	18
packages/repository/src/legacy-juggler-bridge.ts	19	17
packages/context/test/acceptance/class-level-bindings.ts	19	17
packages/context/test/unit/binding.ts	18	17
packages/rest/src/rest-server.ts	15	14
packages/repository/src/decorators/model.ts	15	13
packages/rest/test/acceptance/routing/routing.acceptance.ts	14	13
packages/metadata/src/decorator-factory.ts	13	12
packages/context/src/resolution-session.ts	13	7
packages/repository/test/unit/decorator/model-and-relation.ts	12	11
packages/core/src/server.ts	12	10

Table 3. Comparison of activity in the different packages.

It must be noted that the most commits take place in relatively few files, where only ~64% (394/616) of all files have had more than one commit in `loopback-next`. If we compare this statistic to `loopback` we have ~72% (175/243) of all files that have seen more than one commit.

The amount of commits for the `loopback-next` repository for the last year can be seen in the figure below. What can be seen is that the density of commits have picked up since January 2018.

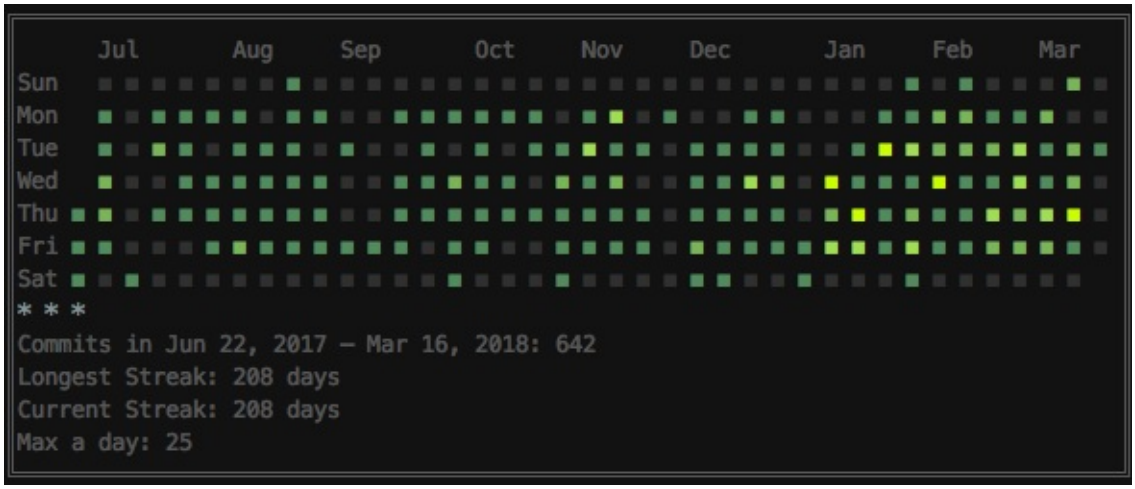


Figure 7. Last year's commits to `loopback-next`

Evolution of code

Since LoopBack4 is in active development, the analysis of the quality of the code of both versions 3 and 4 may be of interest.
As such, we use SonarQube to look into both versions to see how both versions compare to each other. This said, since LoopBack4 is in active development and the quality of code may not be indicative of the final releasable product.

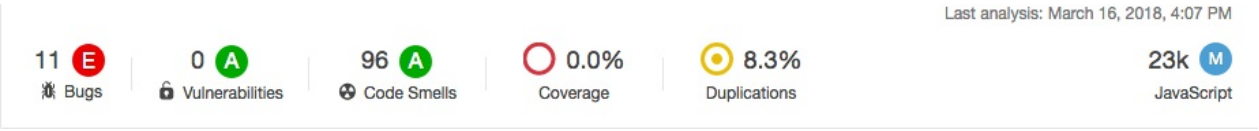


*Figure 8. The quality of the code of LoopBack3.

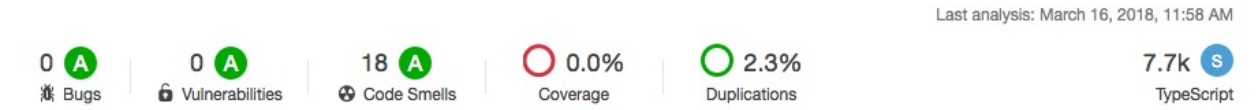


*Figure 9. The quality of the code of LoopBack4.

Since LoopBack4 is a redesign of LoopBack3 there are changes in which the code is organized, which will have an effect on the comparison. When comparing both the quantity of the code is also important to take into account since LoopBack3 contains ~23k lines of code while LoopBack4 only contains ~7.7k lines of code so far. What can be seen is that the rating of the code smells by SonarQube is considered good (*with an A-rating*), with a less than 5 percent as the ratio of the technical debt.

Information viewpoint

The main point of LoopBack is to provide information from a data source to an API. Because LoopBack only provides a way to distribute information it can delegate many information distribution problems to data providers. Issues like physical database separation, large data volumes and backups are also handled by the data source. To make sure LoopBack will not block on any requests which might take a long time in the data source, LoopBack uses asynchronous functions and promises where possible.

Information structure and content

Because the data for LoopBack is very dependent on the application it is hard to define a structure for the information. What we can do is specify some edge cases for which LoopBack should still be able to handle this data.

LoopBack must be able to handle information that is:

- Heavily used
- Changed very often
- Distributed across several data sources

We first need to establish what is considered the information or data LoopBack4 is handling. This is were we run into our first problem; LoopBack4 does not handle a lot of data because it simply provides the tools for the user to build the REST API that handle the data. However, as LoopBack4 provides examples of REST APIs and as the tools do eventually allow the user to handle the data from a database, we will analyze the information flow of a basic REST API created using LoopBack4.

Information flow

As APIs are tools to make information stored in a database accessible to applications, the general data flow is set between an application and a database in both directions (read and write). LoopBack4 provides the tools to create the translation from application requests to the database. As the API created is highly customizable, so are data manipulations the LoopBack4 user wants to put in place. The general idea however is to format the data coming from both sides (application and database) and transport it to the other side.
In order to perform this translation LoopBack4 allows the user to create a connector, this piece of code creates TypeScript functions that map the desired behavior to the database language. It then uses a 3rd party package to connect to the desired database. The connector is part of a DataSource which is a LoopBack identifier for a place where data can be retrieved. Multiple data sources make up a Repository, it is eventually from this repository that the data is accessed by Controllers which manipulate or just present the data to

the application.
Data is converted into Models when traveling through the API. These models are user defined and translated to JSON schemas. At the end the JSON is shown to the application according the OpenAPI specifications. The dataflow is shown in figure 10.

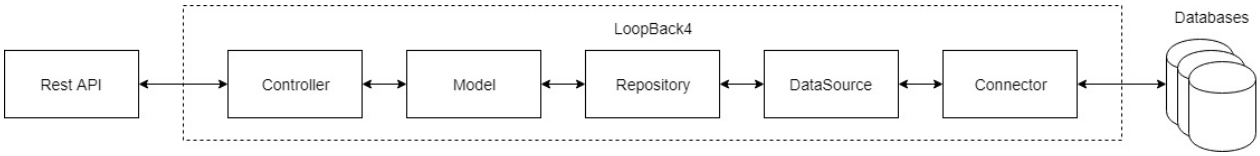


Figure 10. Dataflow of database information from and to the application.

Data quality

Similarly to what has been established before, the data processed by LoopBack4 is only passed through to the application. Furthermore, all functions created by LoopBack4 are designed to be adapted and personalized by the user, therefore the data quality is mainly determined by the user's code.
However, LoopBack4 provides some data quality assessment in the form of promises. The code contains modules that that work with promises meaning that some pieces of code promise a certain data type to other pieces of code. This way the code does not misbehave down stream.

Conclusion

In this chapter we have analyzed the architecture of the open-source project LoopBack. To do this we analyzed the project from different viewpoints. We found that StrongLoop is a company with experienced developers and maintainers who learned from mistakes that were made in LoopBack3. With this new version of LoopBack decisions about package structure and conventions were made beforehand. Because all these structure decisions and conventions are written down, they can be enforced by CI tools and integrators, therefore the `loopback-next` repository is very organized and will probably stay that way. We also found that because the structure and code is very clean the amount of technical debt is quite low for such a large project.

References

Rozanski, N., & Woods, E. (2012). *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley.

Cunningham, W. (1992). The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, *4*(2), 29-30.

conv_commits. https://github.com/strongloop/loopback-next/blob/master/docs/DEVELOPING.md#commit-message-guidelines ↩

commitizen. https://www.npmjs.com/package/commitizen ↩

lerna. https://lernajs.io/ ↩

mocha. https://mochajs.org/ ↩

monorepo. https://github.com/strongloop/loopback-next/blob/master/MONOREPO.md ↩

istanbul. https://istanbul.js.org/ ↩

coveralls. https://coveralls.io/ ↩

about. https://strongloop.com/about/ ↩

ibm_press. https://www-03.ibm.com/press/us/en/pressrelease/47577.wss ↩

planned_features. http://loopback.io/doc/en/lb4/FAQ.html#what-features-are-planned- ↩

Abstract

Mattermost is an open source, cloud-based and self-hosted Slack alternative. The company came to realize the limitations and restrictions of Slack when they adopted it as their messaging service in 2014. This is when they decided to build their own messaging software, Mattermost. In this chapter we analyze the software architecture of Mattermost by looking at the stakeholders, the context view, development view, deployment view, security perspective and technical debt as defined by Rozanksi and Woods [1]. We find out that Mattermost has a well-organized development process and is currently in the process of repaying technical debt by moving the web app to Redux.

Table of Contents

Introduction

The world of team collaboration tools and messaging services has long been dominated by Slack. It has been the dominant service ever since its start in 2014, but the service is set back by limitations due to its proprietary nature and it only available as a Software as a Service (SaaS).

Enter Mattermost, an open source self-hosted Slack alternative. The company behind Mattermost came to realize the limitations and restrictions of Slack when they adopted it as their messaging service in 2014. This is when they decided to build their own messaging software. Mattermost entered the market with the promise of being free from the shackles of privately owned Slack servers. The open source nature makes it accessible for the community to contribute and makes it transparent for clients and the public.

The open source nature of the project has given us the opportunity to analyse and contribute to this project. In this chapter we give our analysis of the Mattermost software by look at views and perspectives as defined by Rozanksi and Woods [1]. We will focus on the webapp portion of Mattermost as this is the part we contributed for. We will look at the stakeholders involved, followed by a context view of all external entities. Next, we look at the development view of the project, which looks at the development process and the architecture of the web app. We look into the technical debt of the project and how it has evolved throughout the development. The final sections take a closer look at the deployment view and security perspective of the project. We conclude our analysis in the final section.

Stakeholders

Various stakeholders are involved in the development of a project. For the Mattermost project we can classify several different stakeholders. The classification scheme is defined by Rozanski and Woods [1].

Stakeholder Classes

Acquirers

The interest of the acquirers is rooted in the value of money the product provides as well as the direction of the company and product. The Mattermost project has several acquirers in the form of investors, which include the following organizations and people: Y Combinator, Rick Morrison, Evan Cheng and Spectrum28 [11].

Assessors

We assume Mattermost Inc., the company who started the Mattermost project, assesses the system's conformance to standards and legal regulations.

Communicators

Product managers communicate a large portion of the concerns of the end users to the designers and engineers. The marketing and sales team communicates with clients and enterprises and are the ones to announce new releases of Mattermost.

Developers

Staff developers and product managers from the Mattermost core team are responsible for the bulk of the development.

Maintainers

The Mattermost project is maintained by staff from Mattermost Inc. and contributors from the Mattermost community. The localization team appoints a maintainer for each language translation, the full list of maintainers can be found in the documentation.

Suppliers

The client is responsible for providing the adequate hardware to operate the system. Mattermost Inc. provides guidelines for hardware requirements and setup.

Support staff

Staff from Mattermost Inc. and community contributors provide support for Mattermost. For enterprise-grade collaborations, Mattermost Inc. provides additional commercial support in the form of Enterprise Edition Support.

System Administrators

The client controls the operation of the system, they are the system administrators of their own deployed system. Mattermost Inc. provides instructions and guides for users who are responsible for running the system.

Testers

The Mattermost core team include QA testers that work closely with staff developers [13]. Mattermost community members can participate in the testing of development branches.

Production Engineers

The Mattermost team does not have designated production engineers, instead this task is relegated to staff developers.

Users

Users are concerned about the functionality and scope of the system and represent the majority of the Mattermost community [14]. Users also include companies and organizations which are in enterprise-grade collaboration with Mattermost [15].

Additional Stakeholders

Contributors

Contributors are part of the Mattermost community and provide contributions for the Mattermost project. Several individuals have been recognized as core committers, a group of trusted individual contributors.

Mattermost Partners

Mattermost provides partner programs to allow third parties to sell the Mattermost system or services related to Mattermost, to customers. Three different partner programs are offered, based on the type of service the third party provides for customers.

- Mattermost Authorized Partner Program, for one-time resale transactions to customers via a fulfillment partner
- Mattermost Value-Added Reseller Program, for on-going resellers who provide support in local language and time zone. Examples of resellers are CounterTrade, Adfinis SyGroup and bytemine.
- Mattermost Deployment Solutions Partner Program, for organizations who can automate the deployment and maintenance process of Mattermost systems. Examples of deployment solution partners are Bitnami, GitLab and AppDome.

A full list of certified Mattermost partners can be found on their website.

Competitors

Mattermost has one main competitor, Slack. Slack is a unique stakeholder; as a competitor it has the power to influence changes to Mattermost, but they can also be an inspiration for new features and ideas. Development of Slack is followed closely to mimic features or keep the quality of Mattermost on par.

Power vs. Interest Grid

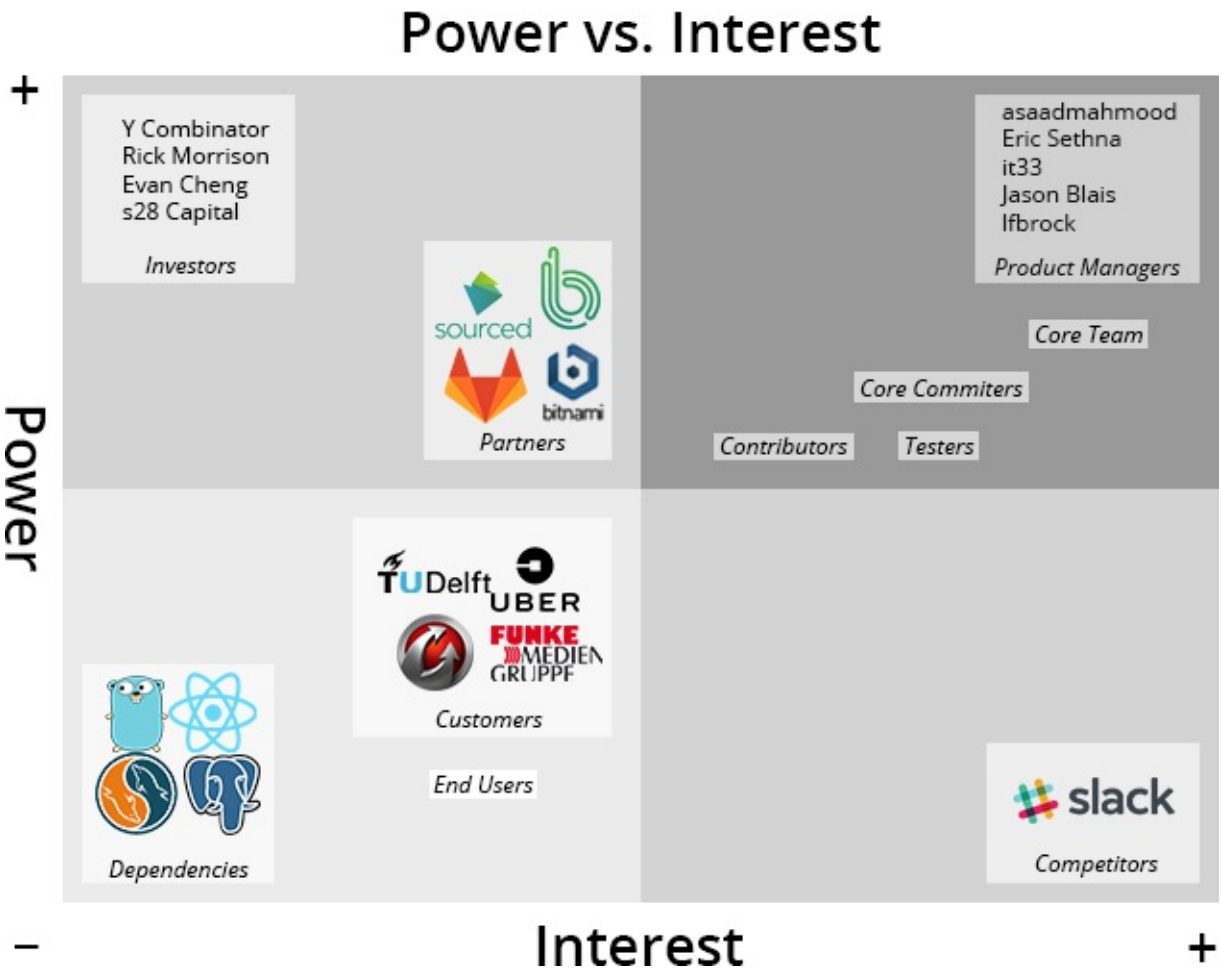


Figure 1 - Power vs. Interest grid for Mattermost

Through the power vs. interest grid we can quantify the involvement of the stakeholders. Power is related to the influence the stakeholder has on the decision making of the system. Interest is related to the interest the stakeholder has on the direction of the system. We can identify how the various stakeholders are involved in this project, by putting them in one of the four quadrants.

At the bottom left quadrant, we observe stakeholders that monitor the system. These include customers, end users and system dependencies. They do not exert much power over the system and only expect the system to function properly.

The top left quadrant contains stakeholders that exert a high amount of power. Investors are necessary for the funding of the project and must be kept satisfied in order for funding to continue. They usually do not meddle with the decision making for the system and thus their interests are low. Partners are in a similar position, but their interests are higher because the system is directly involved in their line of business, thus they must also be kept informed about future developments.

Looking at the top right quadrant, we have stakeholders that manage the system closely. This quadrant contains stakeholders that are involved in the development of the system, such as contributors, core committers, staff developers and the product managers. The product managers have the highest power and interest, their interest in the system is to improve the system; they have the final word on every decision made on the system.

Lastly, we identify the stakeholders in the bottom right quadrant, the stakeholders that keep themselves informed such as competitors like Slack. Slack keep themselves informed about the development of Mattermost, for their systems must be kept on the same level of quality for them to compete. They do not have direct influence on the decision making of the system, their power is low.

Context View

System Scope and Responsibilities

Mattermost has a well-defined scope for their system. During the design process, the Mattermost team discusses what should be part of the scope for the new version. The main scope for Mattermost's current version includes the following capabilities:

- Support one-to-one and group messaging, file sharing and unlimited search history
- Provide native apps for iOS, Android, Windows, Mac, Linux
- Support threaded messaging, emoji and custom emoji
- Provide highly customizable third-party bots, integrations and command line tools
- Support integration via webhooks, APIs, drivers and third-party extensions
- Easily scalable from dozens to hundreds of users
- New improvements released every two months
- Support 14 languages include U.S. English, Chinese (Simplified & Traditional), Dutch, French, German, Italian, Japanese, Korean, Polish, Brazilian Portuguese, Russian, Turkish, and Spanish

Context Diagram

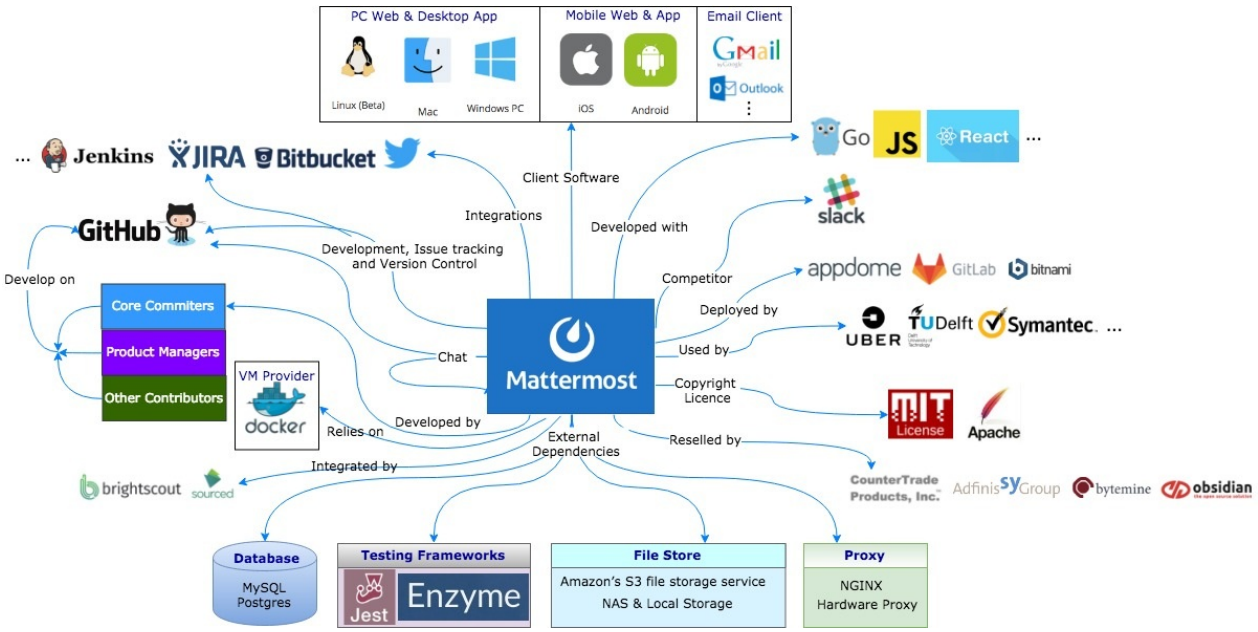


Figure 2 - Context View for Mattermost

External Entities and Interfaces

- The software is offered under either an MIT License or an Apache License v2.0
- Mattermost is used by high profile companies and institutions [11] such as Uber, TU Delft, Symantec, Samsung, Wargaming.net etc.
- Mattermost also has integrations for various services such as Bitbucket, Jira, and Jenkins, etc.
- Mattermost is developed by community contributors, core committers, staff developers and product managers
- Communication is done through their own Contributors Mattermost Channel, forums, GitHub and Jira
- Docker provides a virtual machine for Mattermost servers to run and test on
- Mattermost is mainly developed in Go and ReactJS
- Mattermost client software includes a desktop app, web app and native mobile app
- Mattermost has an email service that uses Gmail, Outlook etc.
- Mattermost uses database systems such as MySQL and PostgreSQL
- NGNIX or hardware proxy servers can be configured for Mattermost servers
- Mattermost supports local file storage, MAS (Mobile App Services) or Amazon S3
- Mattermost uses testing frameworks like Jest and Enzyme
- Partners like Brightscout and Sourced Group provide Mattermost system services
- CounterTrade and Adfinis SyGroup, etc. are resellers of the Mattermost system

- Mattermost works with deployment solution partners such as Bitnami, GitLab and AppDome

Development View

Module Structure

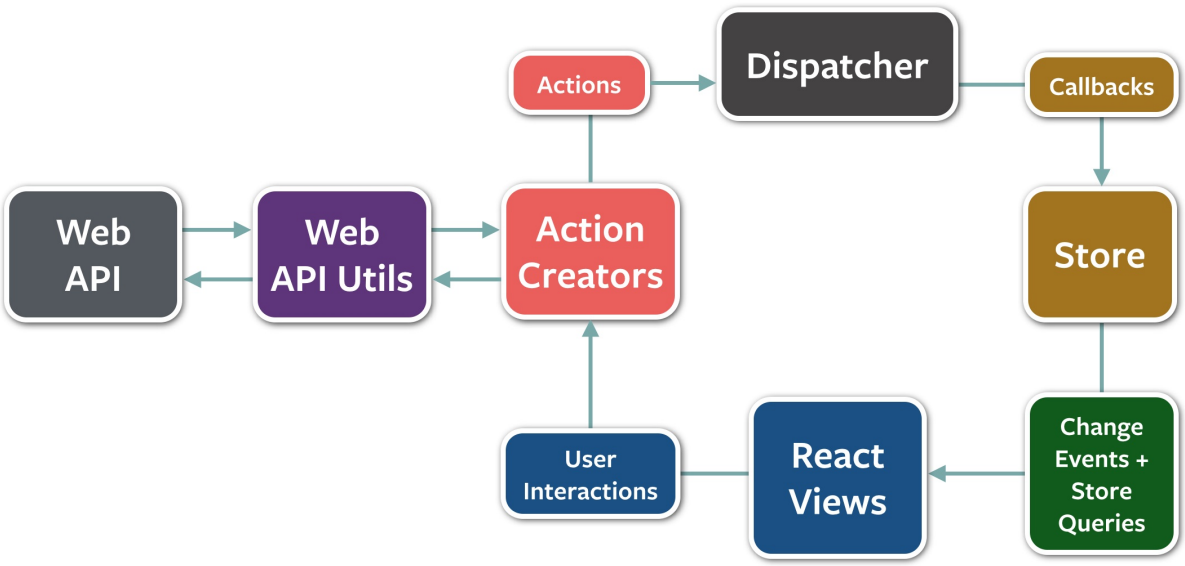
For the module structure we focus our attention on the webapp portion of Mattermost. The webapp is implemented with ReactJS. React is an architecture and an UI library that allows construction of reusable and encapsulated UI components. React is declarative, meaning that components can be built without touching the Document Object Model. The webapp originates from a legacy application that was written in JavaScript and HTML. The webapp uses a react-router library to allow Mattermost to be a single-page web application while still providing the navigation and appearance of a regular website.

The webapp architecture contains several modules which are listed below [6].

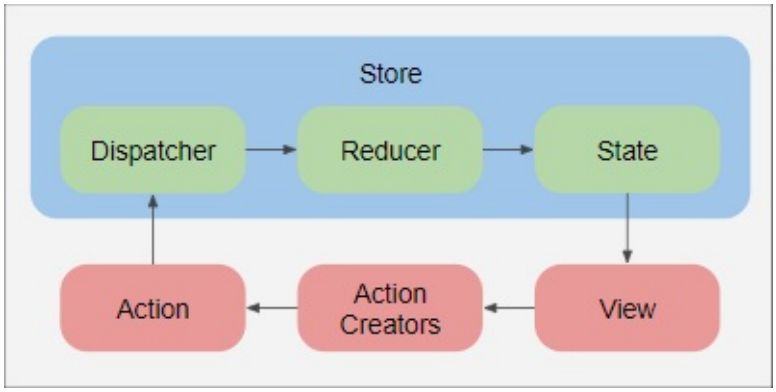
- `components` are React Native components which provide the bulk of the content for the webapp. The React components also define the routes for the React-Router.
- Flux uses multiple `stores` to contain the data used to run the webapp. Redux only uses a single `store`.
- `utils` provides general utility for the webapp, similar to `utils` in the server.
- `actions` are any sort of logic that will result in the manipulation of a store state
- `selectors` are the methods used to retrieve data from the state of the store
- `dispatcher` is used to broadcast payloads to registered callbacks.
- `reducers` take an action and an application state to perform state transitions, this module is unique to Redux.

Flux and Redux

The interaction between the modules is designed with Flux, a pattern that defines how React components performs actions and retrieve data in a one-directional flow. Flux was implemented at a later stage of development, after the webapp was built with React. The interaction between the modules and how Flux is applied is illustrated below [2].



Redux is an library that builds on the Flux pattern which will Flux in its entirety in the near future. While similar, it has a few key differences such as the use of a single store to maintain a single source of truth, and the use of reducers to purely transition between states. These differences are also the greatest benefits of Redux. The webapp is currently being transitioned into Redux, but most of the webapp components such as server interaction and storage interaction are already implemented. The Redux flow and interaction between components as described by Mattermost is shown below.



Release Cycle

The Mattermost system is developed in an iterative way. It is built, tested and released regularly. On the 16th of every month, a new compiled version is released under a MIT license. So far the latest released version is v4.7.2.

Regarding the configuration management, Mattermost is mainly managed on GitHub and the releases are published and controlled on GitHub. New features and bug fixes are tracked and added through Jira tickets. Tickets are prepared for every release, they're put in the pull request queue and are merged when they are approved.

The Mattermost core team has a concrete and unique development process. It draws ideas from Software Development Lifecycle approaches, such as Agile and Scrum. The whole process starts 15 work days prior to the release day until 10 work days after the release day. During the whole process, there are multiple parties from Mattermost involved, among which are the release manager, product manager, developers, logistic teams, quality assurance, marketing team, build team and project leads. Each of them is responsible for different tasks of the release. The following list shows the process of the integrating, testing and building of Mattermost.

- **Integration**: Mattermost does the integration at the beginning of release process. The developers prioritize reviewing, updating, and merging of pull requests for major features 15 days prior to the release day and finish merging the major features 12 days prior to the release date. The whole team will hold meetings to review the status of the remaining pull requests left to merge.
- **Test**: After the cut-off of integration, Mattermost executes a lot of tests to decide which major features will be included in the release. Tests are written and updated in the Release Testing spreadsheet. Tests are mainly done by QA (quality assurance) and are scheduled on 11, 8, 7 and 2 days prior to the release day. These tests are mainly executed in Selenium IDE and the tests results are recorded in the Release Testing spreadsheet.
- **Build**: The building process starts 8 days prior to the release day, when the Release Candidate is cut. The master branch is tagged and branched and "Release Candidate 1" is cut according to the Release Candidate Checklist. Pull requests for bug fixes are reviewed and merged during release candidate testing and new candidate builds are cut accordingly. The CI (Continuous Integration) servers are updated to release branch and the translation server is locked to the release branch. Two days prior to the release day, they tag a new release (e.g. 1.1.0) and run an official build which should be essentially identical to the last release candidate. The release candidates are deleted after the final version is shipped and the release branch is consequently merged into master. The CI and translation servers are updated back to the master.

Internationalization

The release cycle for internationalization or localization is different from the rest of the system. Quality levels are defined for each language [10] to ensure that the final release is of high quality:

- Alpha quality level translations are defined as translations that have not yet reached Beta level.
- For the Beta quality level, 90% of the translations must be verified by a Mattermost expert and a target language expert.
- The Official quality level has all of the translations verified by a Mattermost expert and a target language expert. The target language must also have at least one official reviewer assigned by the Mattermost team who maintains the target language. The target language must also have been in use for at least 3 full release cycles.

Technical Debt

Identifying Technical Debt

In this subsection we identify the technical debt in the Mattermost project. For this analysis we will focus on the webapp portion of Mattermost. We will apply several methods to analyze the technical debt, including the use of automated analysis tools as well as manually looking through the code. Due to the size of the codebase we will only manually look at the code for specific areas. We also look at potential design debts and architectural debts.

Code Debt

Mattermost uses the ESLint linting tool to enforce standards in JavaScript code formatting. The `.eslintrc.json` in the root directory contains the configuration for the ESLint tool. We run ESLint and other code analysis tools through the WebStorm IDE by Jetbrains. After running a code inspection using WebStorm IDE, we observe ESLint reporting 617 errors. Taking a closer look, we discover several code smells.

Firstly, there are issues with code line length. The maximum amount of lines is set to 450 lines, and files such as `utils.jsx`, `sidebar.jsx`, `audit_table.jsx` and `user_settings_notifications.jsx` exceed this limitation. For example, the `utils.jsx` file contains 1329 lines of code, which we can clearly identify as a bloater. Looking through the files, we observe either too many functions in a single file, or functions that are way too long. For example, in `audit_table.jsx` there is a function `formatAuditInfo` which is over 300 lines long. It would be in the developer's best interest to split up the existing long methods into smaller ones for this would improve clarity and scalability.

Next up we observe errors in regards to cyclomatic complexity. The ESLint configuration file sets the maximum tolerable complexity to 10, and there are several functions that violate this threshold. Looking at the violations, we observe functions with a cyclomatic complexity between 11 and 20. These are harder to maintain, but should not be an immediate priority for refactoring or redesign. However, a refactor can definitely make the code more clear and open for future changes.

Most of the rest of the errors are about the liberal use of magic numbers. This is a code smell that the developers are not bothered with, as the violations are very numerous. While refactoring the magic numbers is an option to consider, the sheer number of magic number violations makes it not a straightforward or easy process. We believe that any work put into refactoring magic numbers does not necessarily improve the productivity for future changes.

Design Debt and Impact

To look for design debt we look for design smells. These arise from poor design decisions that make the design fragile and difficult to maintain. Looking at the history of the Mattermost project, we see several periods of time where design debt could have built up. The Mattermost webapp started as a legacy project built on JavaScript and HTML. They first adopted React and later down the line adopted Flux. This decision has caused design debt that is now in the process of being repaid.

From Flux to Redux

The developers are currently working on moving from Flux to Redux. This is an ongoing effort of removing old code and switching to newer code. An example of this is the removal of the old Flux stores and replacing it with a single Redux store.

Pure React components that use Redux get their dependencies passed in through the props attribute, while Flux requires them to be imported. Meaning they no longer have extensive dependency chains, or direct access to data sources. This allows for easier mocking, making unit testing a component in isolation a trivial task. Our contributions to Mattermost included migrating a component to be pure and use Redux. Before the migration the component was not covered by any test, afterwards the component was much easier to test is covered for 86%.

Future Goals

The developers are fully aware of the implications of technical debt and are careful with major refactoring of code. The developers maintain a list of future improvements some of which are related to reducing the technical debt. An example of this is the removal of JQuery, since it doesn't interact well with React and the API. Their main goal with refactoring is to make code easier to test, which in turn should mean less bugs.

Code Coverage Analysis

So far, there are 145 test suites composed of 944 separate tests, targeting on 145 elements, and accordingly 413 Jest snapshots as the standard test output. A simple code coverage run gives us the following table, indicating what percentage of the source code has been covered with the provided test scheme.

Module Coverage Statistics

Modules	Files Covered	Lines Covered
`selectors`	5	54%
`reducers`	8	53%
`utils`	24	43%
`plugins`	3	30%
`stores`	17	27%
`client`	2	16%
`actions`	23	13%
`components`	N/A	N/A

Redux `reducers` and `selectors` are the most well-covered modules of the project. They are well tested mainly because of the simplicity of Redux, and that currently the team is going through the process of moving from Flux to Redux, thus all newly wrote components having corresponding test suites.

On the contrary, the other modules are poorly tested, much less lines covered, because they are either rather complicated or old, comparing with newly evolving Redux modules. Take the module `actions` for example, it holds all Flux actions where the majority of the logic of the webapp takes place. The same goes for `stores` which holds all the Flux stores. The low coverage shows that there is a large debt that is being repaid by moving to Redux.

Component Coverage

As for the `components` module, it should be broken down further to talk about the statistics, given that there are currently hundreds of components. About half of the components are fully tested, with a coverage over 85%, and about one third of the components are not tested at all. This has to do with the transition to Redux.

Evolution of Technical Debt

Mattermost is concerned with technical debt and has shown to put in effort to repay this debt. The origin of Mattermost has a play in how the technical debt build up. Mattermost used to go by the name SpinPunch, a HTML5 game studio. The first version was not designed to be a Slack alternative but a way to reach gamers. Due to internal problems with proprietary messaging apps they planned to use it for their own company communications.

The webapp and server were not originally designed to support the features it currently has. The webapp was built on a legacy application using JavaScript and HTML. In addition, the webapp and the server were initially not separated, but part of the same repository. It was only later that these two portions were separated. This in turn reduced the technical debt for both the server and the

webapp, as the separation made both aspects easier to maintain. This also meant that certain parts had to be completely rewritten to maintain a simple and effective architecture. In 2015 the webapp was rewritten using the Flux pattern. The Flux pattern has a few advantages, namely:

- A clear structure
- A clear data flow
- Easy to add new components
- Per component testing

This was a step in the right direction, however to further simplify the webapp a different method could be used: Redux. The main difference being that Redux uses one store, while Flux uses multiple stores. Transitioning has started, and a lot of components are now pure Redux components. This transition has been divided into smaller tasks to receive aid from the community. Smaller tasks have the additional benefit of avoiding big code changes which could cause problems for people maintaining a fork. This prevents technical debt from building up and allows a smoother transition from Flux to Redux.

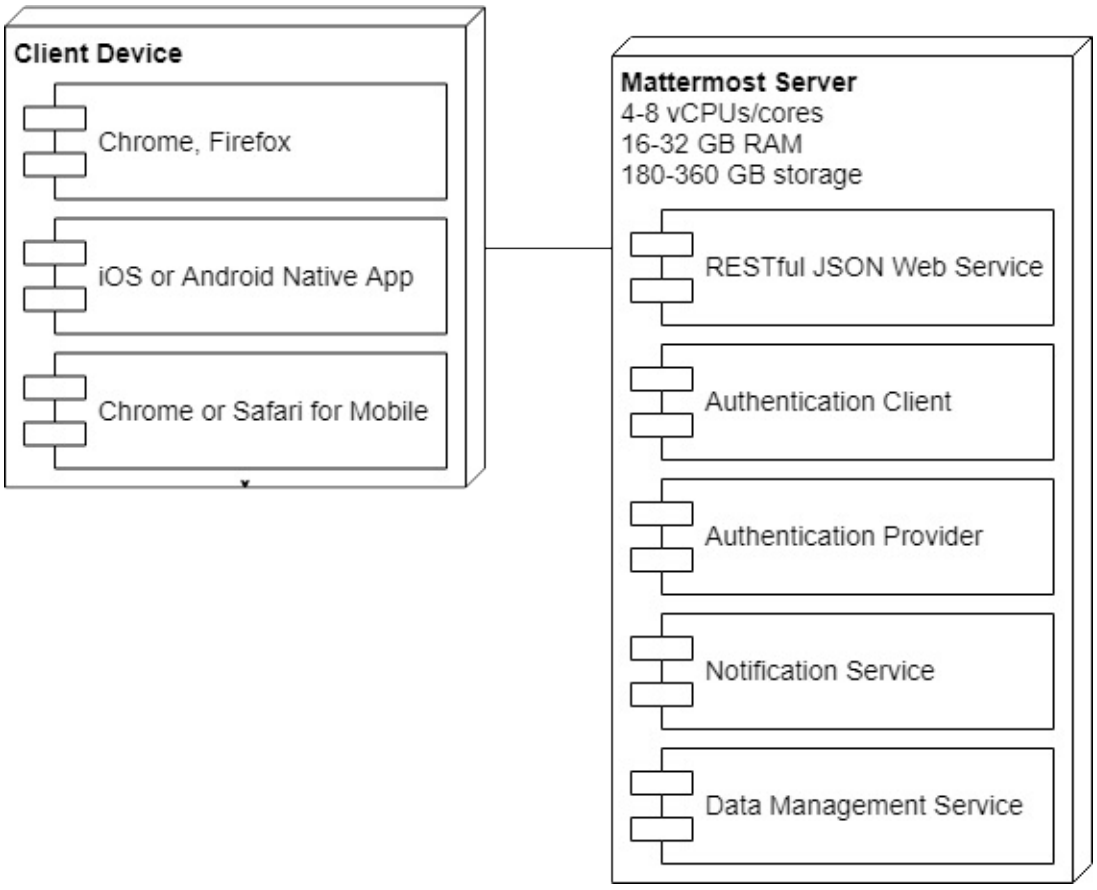
Deployment View

The deployment view provides an overview on aspects of the system after the system is ready for deployment. It defines the hardware the system needs to operate, as well as the network requirements, and the software that is mapped to the various runtime elements [1].

Runtime Platform Model

The runtime platform model is the core of the deployment view. We provide the runtime platform model for the team edition as well as the enterprise edition.

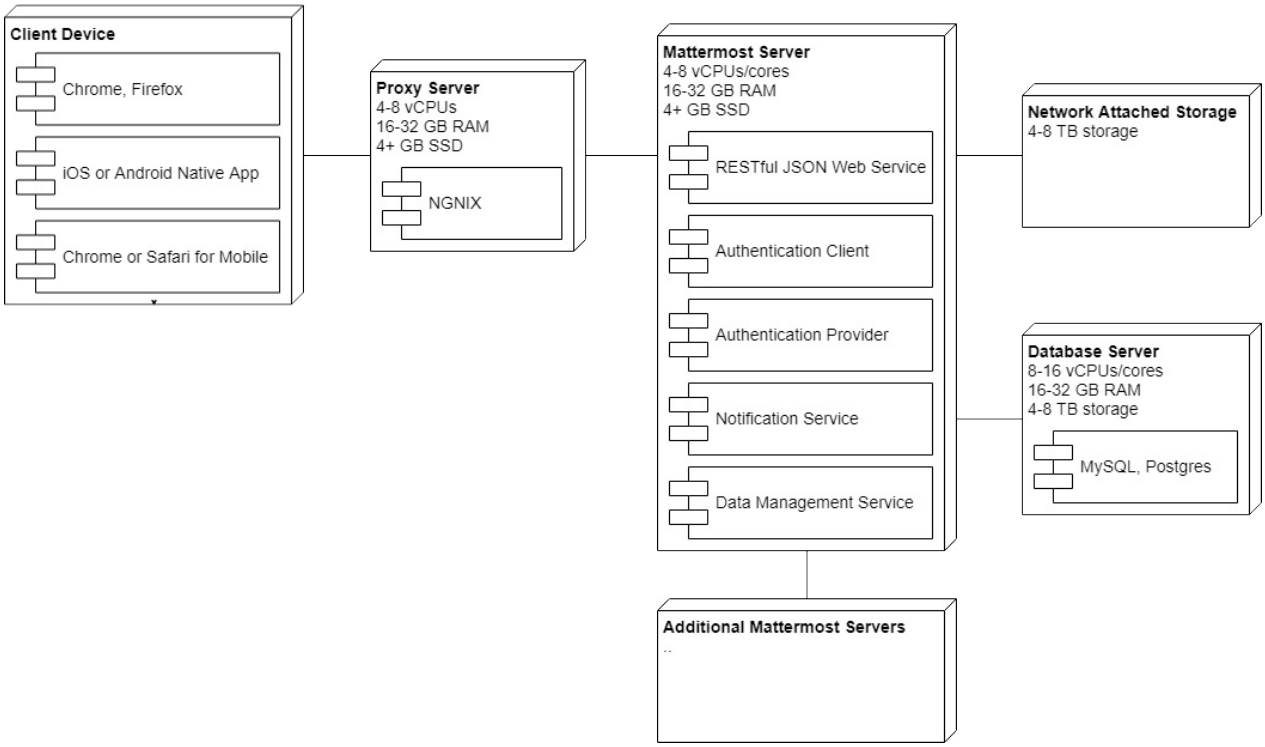
Mattermost Team Edition



Note that the optimal configuration for the Mattermost Server can vary by the amount of registered users. The runtime platform displays the setup for 1000 to 2000 users. Other set ups are found in the documentation.

Only a single machine is needed to deploy the team edition, cutting down on deployment costs and complexity.

Mattermost Enterprise Edition



The Enterprise edition has its hardware set up for high availability and scalability. The hardware needed for the Enterprise edition is beefy and provides capacity for up to 20000 registered users and a peak of up to 4000 concurrent users.

For high availability mode, additional servers can be added. High availability mode provides better concurrency and provides more options for back-up and recovery.

Technology Dependency Models

Software Dependencies for Client Devices

Component	Requires
PC Web	Windows 7, Windows 8, Windows 10 with IE 11+, Chrome 43+, Firefox 52+ or Edge 40+
Mobile App	iOS 9+, Android 5+
Mobile Web	iOS 9+ with Safari or Chrome 43+, Android 5+ with Chrome 43+

Software Dependencies for Mattermost Server

Component	Requires
Mattermost Server	Ubuntu 14.04, Ubuntu 16.04, Debian Jessie, CentOS 6.6+, CentOS 7.1+, RedHat Enterprise Linux 6.6+, RedHat Enterprise Linux 7.1+, Oracle Linux 6.6+, Oracle Linux 7.1+
Docker	Docker Compose

Software Dependencies for Database Server

Component	Requires
Software	MySQL 5.6+, PostgreSQL 9.4+

Security Perspective

Security is a top concern when it comes to a messaging service, given that it deals with tons of user information. It is even more so for team messaging services, like Mattermost, handling the internal communication of an organization or a company, which may involve commercial secrets worthy of millions. Security matters at Mattermost and we decide to take a look at the project from this perspective.

Sensitive Resources

Almost everything is considered as sensitive, from information like login details to operations like inviting new members. Here we give a table of resources we think important and the reason why.

Resource	Sensitivity	Owner/Operator
Login credentials	Information to identify a user. May result in identity theft or privacy invasion.	The user
Chat history and uploaded files	Might contains commercial secret or user privacy.	The sender and receiver
Info of channels/teams	Contains information about the structure or hierarchy of the organization.	The organization
Editing a post	Needs to be controlled to protect the integrity	The sender
Modifying user details	Needs to be controlled to prevent malicious changes	The user
Inviting users to a private channel/team	Needs to be controlled to prevent information breach or spamming	Admins
Modifying info of channels/teams	Needs to be controlled to prevent malicious changes	Channel/team admins
Entering system console	Needs to be controlled to prevent malicious operations	Admins
...	...	...

There is much more to be listed, like the using habit of a user can be exploited to impersonate him or invade his privacy. The interest of stakeholders can be harmed from various angles by compromising sensitive resources.

Security Policy

Mattermost secures things by categorize users into different roles. The following table displays what these roles are and their permission levels.

Roles and Permissions

	Posting	Invite/Delete Channel Member	Assign/Remove Channel Role	Edit/Delete Channel	Invite/Delete Team Member	Assign/Remove Team Role	Edit/Delete Team	Access System Console	Deactivate/ Reactivate User	Assign/Remove System Role
Member	Yes									
Channel Admin	Yes									
Team Admin	Yes									
System Admin	Yes									
Deactivated User										
Random guy outside the server										

Admins can set the channel/team open to invitation, otherwise only admins can invite new members. Users can be deleted from a channel or a team but not from the server. They can only be deactivated to prevent compromising the integrity of message archives from happening. And the Admin roles are specific for that specific Channel or Team, meaning that one Team Admin might be nobody in another team. The actual access permission model is more complicated than the table because System Admin can manage separate permissions from anyone. It is a hierarchy structure with roles at higher positions also have the permission of lower position roles.

Security Implementation

One thing worthy of mentioning before walking down the security implementation of Mattermost is that, the server of Mattermost is deployed on the organization's local network, behind the private firewall. So a large part of the responsibility of security falls onto the organization itself.

Besides the shield of the private firewall, Mattermost also performs a few practices to ensure the security of the service. It supports TLS encryption using AES-256 with 2048-bit RSA on all data transmissions between Mattermost client applications and the Mattermost server across both LAN and internet, to prevent eavesdropping. Encryption-at-rest is available for messages via hardware and software disk encryption solutions applied to the Mattermost database. And to protect against brute force DDoS attacks, the organization can set rate limiting on APIs, varied by query frequency, memory store size, remote address and headers. Session length, session cache and idle timeout are all configurable to comply with the internal policies of the organization.

As for recovery, the Mattermost team has published a guideline for administrators to tackle backup and disaster recovery. It provides a `High Availability Mode` while deploying to allow fast automated recovery from a component failure.

Evaluation from the Security Perspective

Though Mattermost touts its concern for privacy and safety as a selling point against its competitors, there are a few things of the project, that are questionable.

For example, to enable end user search and compliance reporting of message histories, Mattermost does not offer encryption within the database. Mattermost also stores a complete history of messages, including edits and deletes, along with all files uploaded. Even if the user deletes a post or a file, it only disappears from the interface but stays in the database. Though it is understandable from the perspective of protecting integrity and non-deniability, it can instead be a problem for user privacy.

Another problem of Mattermost is that it depends too much on the private network. For example, the team made a deliberate decision to show "Your password is incorrect" when wrong password is entered, which can be considered as a information leakage and can be exploited by the attacker. The team argued that if security was of top concern, then the service should be deployed on a private network.

Conclusions

Our journey with the Mattermost team is extremely pleasant as they are very community friendly. Deeply involved with the community, that's how they maintain the momentum and prosperity as a small team, and on the other hand, that's also a big challenge. Luckily the team has established an effective way of developing and managing the project. The project is well organized as they have a clear roadmap, of how the structure of the project is supposed to be, in mind. Though some debts were made during the earlier days of Mattermost, they are acting actively to repay them by moving from Flux to Redux. With people's growing awareness of privacy protection, Mattermost is surely going to earn a larger part of the pie with its ability to deploy the server locally.

References

[1] N. Rozanski & E. Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, 2012.
[2] "facebook/flux: Application Architecture for Building User Interfaces." Internet: https://github.com/facebook/flux [Mar. 5, 2018]
[3] "Mattermost Dev Talk - Introduction to Codebase & Architecture." Internet: https://youtu.be/Q4MgnxbpZas [Mar. 5, 2018]
[4] "Mattermost Dev Talk - Server Application Layer." Internet: https://youtu.be/rU6-xSV6dTI [Mar. 5, 2018]
[5] "Mattermost Dev Talk - Localization." Internet: https://youtu.be/cVRmzXjpp7Y [Mar. 5, 2018]
[6] "Mattermost Dev Talk - ReactJS." Internet: https://youtu.be/8dB2_29TtNo) [Mar. 5, 2018]
[7] "Mattermost Dev Talk - Building a Webapp Component." Internet: https://youtu.be/MRmGDhlMhNA [Mar. 5, 2018]
[8] "Mattermost Dev Talk - Adding a Config Setting." Internet: https://youtu.be/hVLRVPnNeaw [Mar. 5, 2018]
[9] "Release Process — Mattermost 4.7 documentation" Internet: https://docs.mattermost.com/process/release-process.html [Mar. 5, 2018]
[10] "Localization — Mattermost 4.7 documentation" Internet: https://docs.mattermost.com/developer/localization.html [Mar. 9, 2018]
[11] "Company - Mattermost." Internet: https://about.mattermost.com/company/
[12] "Contributors to mattermost/docs." Internet: https://github.com/mattermost/docs/graphs/contributors
[13] "Staff Developers — Mattermost 4.8 documentation." Internet: https://docs.mattermost.com/process/developer.html
[14] "Mattermost Community — Mattermost 4.8 documentation." Internet: https://docs.mattermost.com/process/community-overview.html
[15] "Success Stories – Mattermost." Internet: https://about.mattermost.com/success-stories/

Mbed OS

By Jasper de Winkel, Arjan Langerak, Wanning Yang and Kun Jiang.

On Github jdewinkel, alangerak, yangwanning and jiangkun1994

Abstract

Mbed OS is an open-source embedded operating system designed for the "things" for the Internet of Things. It supports a lot of different target hardware as well as the option to implement support for other hardware without altering the existing libraries. This chapter analyzes Mbed OS by starting with a discussing of the stakeholders and the context of the system. After that, the development is discussed which describes the architecture of the code. Furthermore, the deployment of Mbed OS to different target hardware is discussed . Then, the evolution of the system and the technical debt of the system is discussed. This chapter then ends with a conclusion on the status of the architecture.

Table of Contents

Introduction

ARM Mbed is a fully integrated device management solution. There are some services it can provide, such as operating system, gateway, device management services and partner ecosystem [1]. These services may reduce costly development and deployment time of IoT solutions. The solution is divided into two major parts Mbed Cloud and Mbed OS. This document focuses on Mbed OS specifically.

Mbed OS

ARM Mbed OS is a free, open-source embedded operating system designed specifically for the "things" in the Internet of Things. It includes all the features needed to develop a connected product based on an ARM Cortex-M microcontroller, including security, connectivity, a RTOS and drivers for sensors and I/O devices.[3].

The RTOS is the major compoment in Mbed. This is a type of OS that is intended to handle tasks that are defined with a minimum and a maximum time in which they should have been executed. This is achieved by using a combination of threads and a scheduler that determines the order of the tasks.

Stakeholders

A lot of different stakeholders have interaction with Mbed OS. These stakeholders are separated into different categories and described in the table below.

Table 1. The different stakeholders of Mbed OS

Stakeholder	Description
Acquirers	The enterprise ARM takes the main responsibility of developing and marketing Mbed operating system to cooperative partners. These partners contribute to the platform, use it within their products or deliver a service to Mbed. The OS directly targets ARM processors and supports different microcontrollers of vendors like NXP and ST. These vendors are also ARM Partners and act as acquirers because they actively spend resources on developing support for MbedOS to get their boards working with MbedOS.
Assessors	Assessors determine if the legal constraints of the project are met. In the case of Mbed, this is the responsibility of ARM as it is supporting and pushing Mbed.
Communicators	ARM handles communication through (paid) support, a news letter and by support forums. From the releases it is deduced that one key communicator (ARM employee) is @adbridge.
Developers	In the case of Mbed, the most active developers are a part of ARM or one of its partners. Key developers are:@bcostm, @0xc0170, @theotherjimmy, @geky and @pan-
Support staff	Support of MbedOS is given in two ways. Number one is the commercial channel. Since it is a commercial product owned by ARM, support can be provided directly. The other support channels are through the forums and questions can be answered by email.
Suppliers	Key suppliers are the compilers and libraries. Relevant compilers are the ARM compiler or GNU ARM Embedded. The code is hosted on Github, thus Github is a supplier as well.
Users	Since MbedOS targets an embedded device the responsibility of deploying, designing the hardware and software environments of the system is a task for the end user. In this case, an embedded developer working at a company. Maintenance of the final product also will be the responsibility of the embedded developer.
Contributors	Some contributors from ARM are also integrators while there are still some contributors are not from ARM. @bridadan, @emilmont and @stevew817 are three of these active contributors.
Integrator and Maintainers	In Mbed OS no clear distinction is made between integrators and maintainers. The integraters take on both roles. The main integrators at the moment are @0xc0170, @geky, @pan-, @theotherjimmy and @cmonr. Most of the integrators are ARM employees.
Competitors	MbedOS is not the only platform, it has many competitors. Several compatitores are: Arduino, Tizen, Predix and FreeRTOS
Founders/Originators	The Mbed project began in 2005 when the two founders Chris Styles and Simon Ford (ARM employees) met to discuss some projects they had been helping out with[17].

Power-Interest Grid

The grid listed below shows the power vs Interest of each stakeholder. ARM as the major acquirerer has the highest power and is the most invested in the platform. With slightly less power but with also very large interest are the Integrators/Maintainers followed by the developers. The development of Mbed OS is closely monitored by the competition. The communicators and users have moderate

interest in the platform but have little influence. Suppliers however have a moderate to large power since if there would be any major changes or a stop of development this would directly effect the project. However, programming languages like C and compilers like GNU ARM Embedded are very unlikely to change drastically.

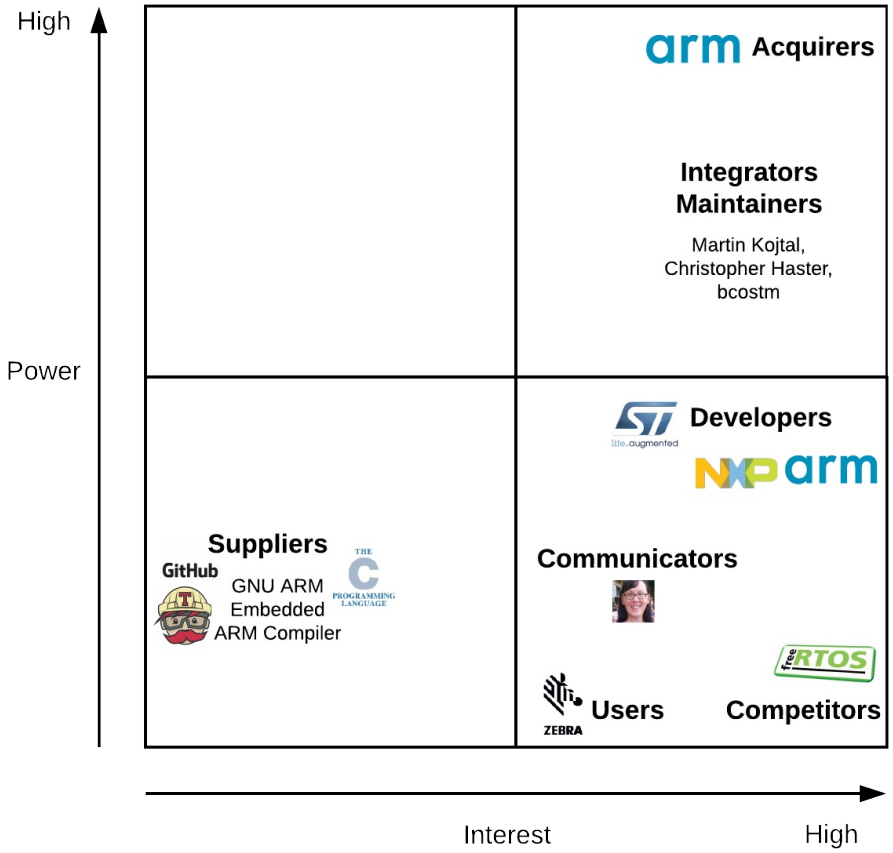


Figure 1. Stakeholder power vs interest grid

Context View

To create this large ecosystem, Mbed depends on a lot of different parties that are not part of Mbed (or ARM, the parent company). These parties can be directly involved with the development of the OS while others provide online services to manage IoT devices that are running Mbed. Below are some major interactions and dependencies listed and visualized. Not all the companies that provides software,hardware or services are listed since there are too many to be able to include.

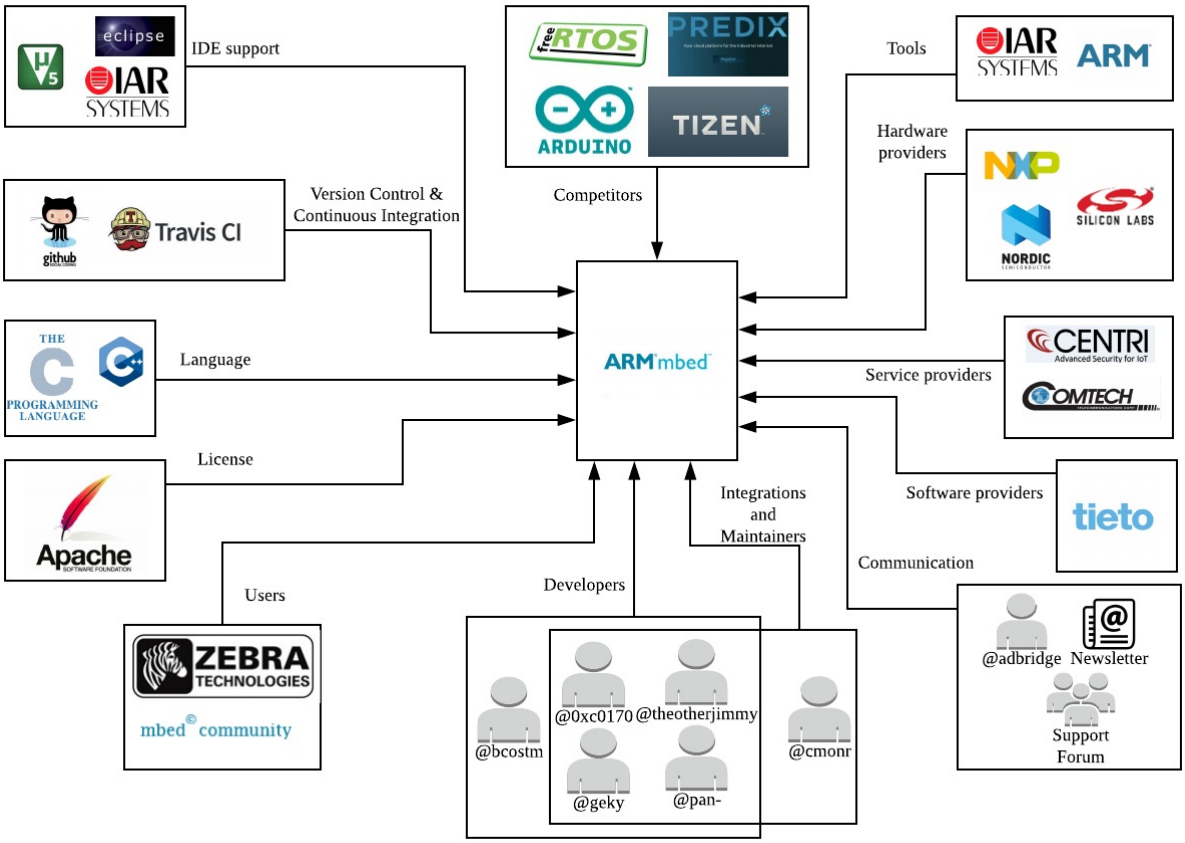


Figure 2. Context View of Mbed with its environment

Following up on the figure, a more detailed description is discussed next.

Programming Language: The bulk of the software is written in C. But to give users a more friendly way using this, a small C++ layer is added. So, the C code can be seen as the External Entity while the C++ fulfills the role of a *Service Provider* for the External Interface.

Version Control & Continous Integration: Several Github repositories are used for Mbed, one of which is specifically for Mbed OS. It is used for version control and issue tracking. Besides this, they also have a repository on their own website[16] for sharing community project and example projects.
On all Pull Requests, test code is run by using Continuous Integration tools. In order to integrate Travis CI with Mbed and Github, an *External Interface* exists that provides an API to achieve this. So, this is a *Service Provider*.

IDE support: A project can be set up on major Embedded Systems IDEs like Keil uVision5 but Eclipse is also supported. This tool is a *Data Provider* because it created to necessary data to be able to use the IDE.

Hardware Providers: These companies provides hardware components for IoT platforms.

Service Providers: These offers *External Interfaces* to be able establish a connection between a board and their services. These interfaces can be classified as *Data Providers* (data storage), *Service Provider* (security) and *Event Provider* (monitoring services).

Software Providers: Tieto provides several protocol stacks and drivers. Since this software is directly included into Mbed, there is no *External Interface* needed.

Development View

The development view “describes the architecture that supports the software development process” [4]. In this section, the structure of MbedOS is introduced in terms of the architecture of the source code, standardizations and the structure of the directories.

Module structure

MbedOS is a layered architecture. At the bottom, the specific hardware drivers of a vendor can be found. Then, each layer that comes after this adds either new functionality or simplifies the usage of that device by providing a common API.

The system can be roughly divided into three sections: the *Core* , *Storage* and *Connectivity*.

The *Core* consists out of modules that forms the Real Time Operating System. Also, the modules for interfacing with peripherals and for creating a sandbox environment (uVisor) are part of the *Core*. The reason for this classification is that these modules provides the functionality to run code on a hardware board. Therefore, all the other modules have dependencies on modules that are part of the *Core*.

The different modules that are related to files and storage are grouped into the *Storage* section. The *FileSystem* and *Storage Managment* provides APIs for interfacing with regular files. The *NVStore* provides the functionality to store data by keys in internal flash.

Lastly, all the different modules that are related to connecting devices together are found in the *Connectivity*. Mbed has support for Wifi, Bluetooth and also Low-Rate Wireless Personal Area Networks (LR-WPAN). The *Thread* , *6LOWPAN* and *nanostack* are modules for building a complete network between different devices. In particular, *6LOWPAN* provides IPv6 functionality and *Thread* provides functionality to create a mesh network.

The categorized modules be found in the figure below.

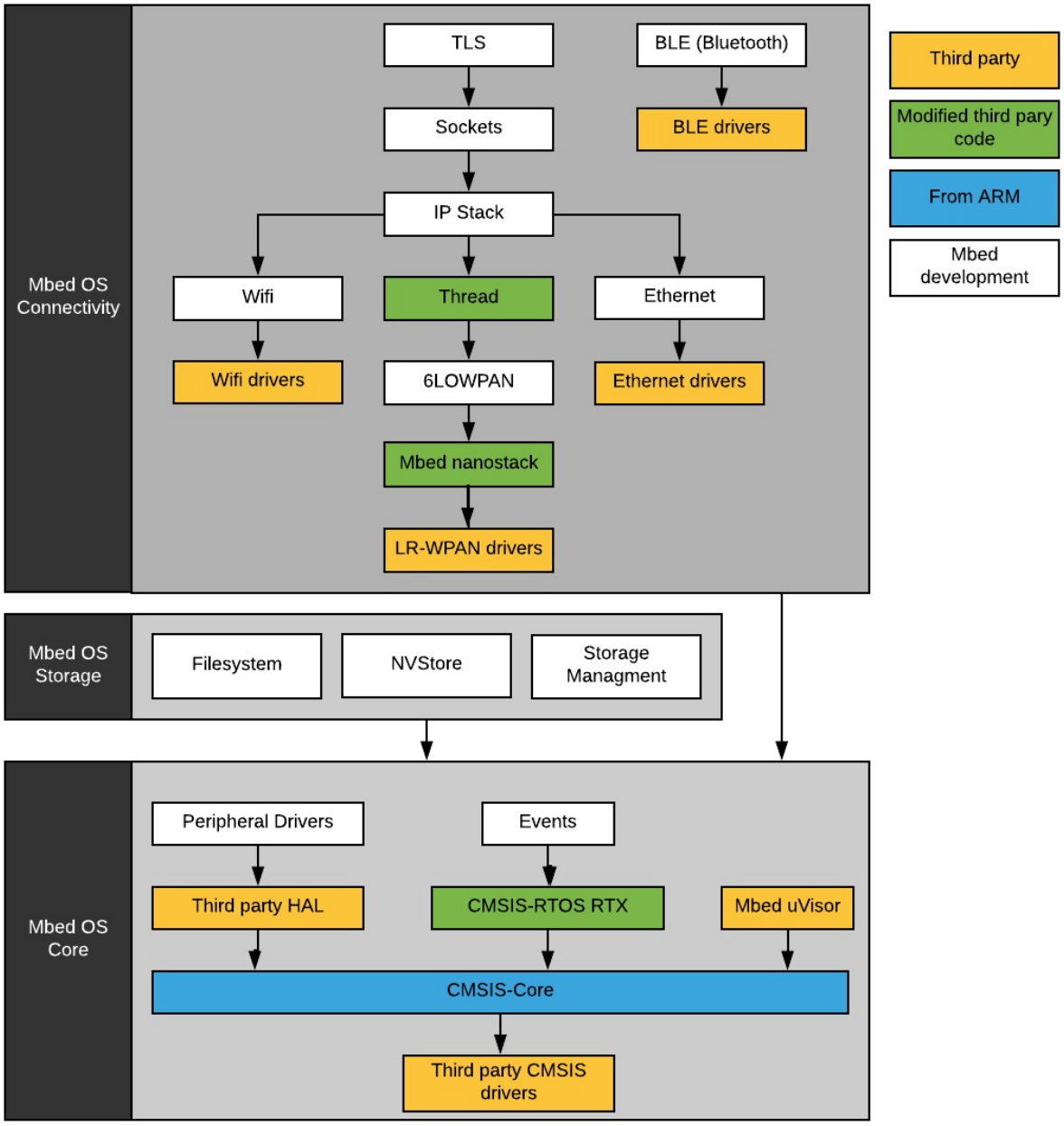


Figure 3. Module overview of the most important components of the system.

Common Design Models

Contributing standardization

Open source projects such as Mbed OS are developed by both employers of Mbed as well as community members. This means that everyone is free to contribute to the repository. In order to keep maintainability, reliability, and technical cohesion of the system, the core developer composed a *contributing and publishing guide* to standardize the design process. It provides concrete guidelines in terms of the code, naming conventions, documentations and compiler settings which can be all found on their style guide page[5].

API design standardization

In addition, the API design should be worth mentioned here as it provides an environment to implement groups of features like drivers, connectivity etc. The general rule is that a C++ class-based interface is implemented for MbedOS users. Another one is the use of a porting layer which handles the support of multiple hardware targets. This layer is implemented with a C compatible interface.

This API design is splitting into the *drivers* directory (C++ interface), the *hal* (C interface) and the *targets* (implementations) directory.

Also, this API uses a configuration file that defines what kind in inputs/outputs a board which is shown below.

```
"LPC11C24": {
        ...
        "device_has": ["ANALOGIN", "CAN", "I2C", "I2CSLAVE", "INTERRUPTIN", "PORTIN", "PORTINOUT", "PORTOUT", "PWMOUT", "SERIA
L",
        ...
    },
```

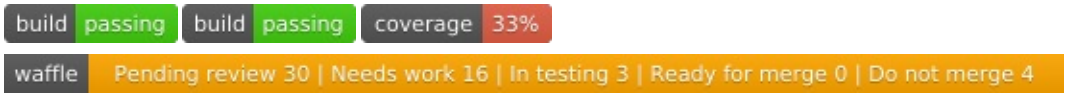
By using *if* and *endif*, the corresponding drivers can be included into the build process.

Testing standardization

Since the MbedOS accepts community contributions through GitHub, it requires some form of automated testing. To accomplish this, Mbed uses Travis CI (continuous integration) and it is ran on all pull requests and all commits to a branch to verify that it builds and it passes testing. Regardless of which type of CI run for testing, all CI jobs must pass before the pull request is merged by Mbed OS maintainers.

The process starts at the *.travis.yml* file in the root of Mbed OS repository. It specifies all the steps for building a test environment. The build lifecycle is made up of two steps, one is **install** which installs any dependencies required and the other one is **script** which runs the actual tests. In *script*, multiple script commands are specified. If one of the build commands returns a non-zero exit code, the Travis CI build runs the subsequent commands as well, and accumulates the build result.

Finally, the results of testing are passed to *coveralls* which reports this as shown below.



Since Mbed OS is mainly working for different kinds of boards, it should have a standardization testing for functionality of drivers. Mbed provides the *Greentea* testing tool to automate the process of flashing Mbed boards, driving the tests and accumulating test results into test reports. Those all *.cpp* files in 'mbed-os/TESTS/mbed_drivers' directory indicate that which certain drivers need to be tested. For example, flash IAP, low-power Ticker, Real-time Clock, etc. Each testing file will include a corresponding testing process to guarantee the executable driver and finally can meet the high demand of real time.

Generally, there are some common parts in testing files. Firstly, the following code can be found in all *.cpp* files which introduces the used testing tools.

```
#include "mbed.h"
#include "greentea-client/test_env.h"
#include "utest/utest.h"
#include "unity/unity.h"
```

And there is also a list of specified cases in each file for testing certain functionality. Here, we select the *main.cpp* of *lp_ticker* as an example. The low power ticker, mainly working on setting up a recurring interrupt and calling a function repeatedly at a specified rate. Therefore, the test will include measuring the callback time and checking whether ticker properly execute callback. The code of this part is shown as following:

```
// Test cases
Case cases[] = {
    Case("Test attach for 0.001s and time measure", test_attach_time<1000>),
    Case("Test attach_us for 1ms and time measure", test_attach_us_time<1000>),
    ...
};
```

Greentea also presents relevant testing requirements in order to match the standards. For example, tests should be organized based on the class, roughly one test file per class.[7]

Codeline

Figure 4. The top level directory structure of the code [9]

The figure above shows the top level directory with the smaller directories already expanded. These directories will be explained first.

The cmsis (vendor-independent hardware abstraction layer) contains abstract interfaces of the various hardware components found in CORTEX microcontrollers.

The driver directory contains C++ interfaces that provides abstractions for common features found in boards. These includes interfaces for input/output pins and other common hardware.

The events directory includes a third party library called *equeue* written in C which provides event scheduling by using queues. Mbed has encapsulated this library with C++ classes.

The hal (hardware abstraction layer) includes C interfaces that defines functions for interacting with hardware. Some of these interfaces are used in the *drivers* directory that provides the C++ encapsulation.

The platform directory includes C++ classes that implements basic functionality that are commonly found on different boards and platforms.

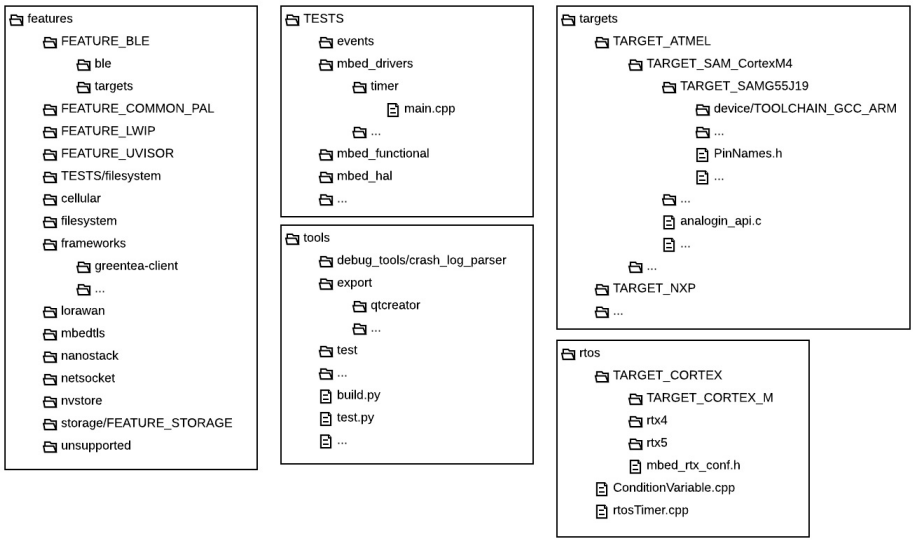


Figure 5. Directory structures of features,TESTS,tools,targets and rtos.

The features directories contains all the different libraries for using wireless communications. Also, libraries for storage and the sandbox (UVisor) are found here. The common structure that is found between these libraries is that there is the code that controls the hardware is separated from the protocols and algorithm code. The hardware code is always found in the *targets* directory.

All the tests that are targeted Mbed are found in the TESTS directory. Each module has its own directory inside the TESTS directory. These modules also have their own directories where the names reflects what part is being tested. Then, the actual testing file is always called *main.cpp*.

The tools directory contains the different python script that can generate IDE config files, run tests, build source code and other various scripts that are used for configuring,building and uploading code to boards.

The targets directory contains the files that implements the interfaces found in the *hal* directory. Also, the initialization files and interfaces from the *rtos* directory are also included. The structure of this directory is that the boards of each manufacturer are grouped together. The manufacturer then has defined their own structure of how to seperate different boards.

Finally, the structure of the rtos directory is that the top level contains basic C++ classes that encapsulates some functionally of the underlying rtos which is written in c. The *TARGET_CORTEX* directory contains the directories where the actual rtos is located, Keil RTX rtos. Also, the *TARGET_CORTEX_M* contains files that are related to error handling on the Cortex M microcontrollers.

Deployment View

With the large amount of supported targets and options, Mbed OS's build and deployment is one of the most challenging aspects of the OS. As mentioned in previous sections, Mbed has its own python builder to set specific flags in code to enable certain drivers/modules depending on the desired features and the hardware target. The table that highlights a few major third party dependencies for deployment.

Table 2. Third party dependencies for deployment

Category	Third party library	Mandatory	Role in the system
Building:	GNU Arm Embedded Toolchain	Choose one	Compiler, C/C++ libraries etc. to build the OS
	ARM Compiler and Toolchain	Choose one	Compiler, C/C++ libraries etc. to build the OS
Processor abstraction:	CMSIS	Yes	Cortex Microcontroller Software Interface Standard
OS functionality:	Keil RTX	Yes	Real Time Operating System (RTOS) that manages scheduling of tasks, semaphores etc.
Event handling:	Equeue	Yes	Provides event scheduling by using queues
Networking:	lwIP	No	Small lightweight TCP/IP stack
	LoRaWAN	No	Provides support for LoRa
File System	Cfstore	No	Secure, associative key-value (KV) store abstraction layer
	ChaN	No	FatFS implementation
Security:	MBed TLS	No	Embedded SSL/TLS library
Testing:	GreenTea	No	Mbed's own testing platform
	Utest	No	Embedded Testing
	Unity	No	Unit Testing for C (especially Embedded Software)

Mbed OS is a RTOS thus requires a user program to tell it what to do in order to do anything. An example deployment scenario is shown below. It depicts an embedded product with an onboard microcontroller and some other hardware on the board for communication. It can communicate directly to other devices through Mbed by using simple peripherals or even by using the on-board ethernet hardware to connect to servers.

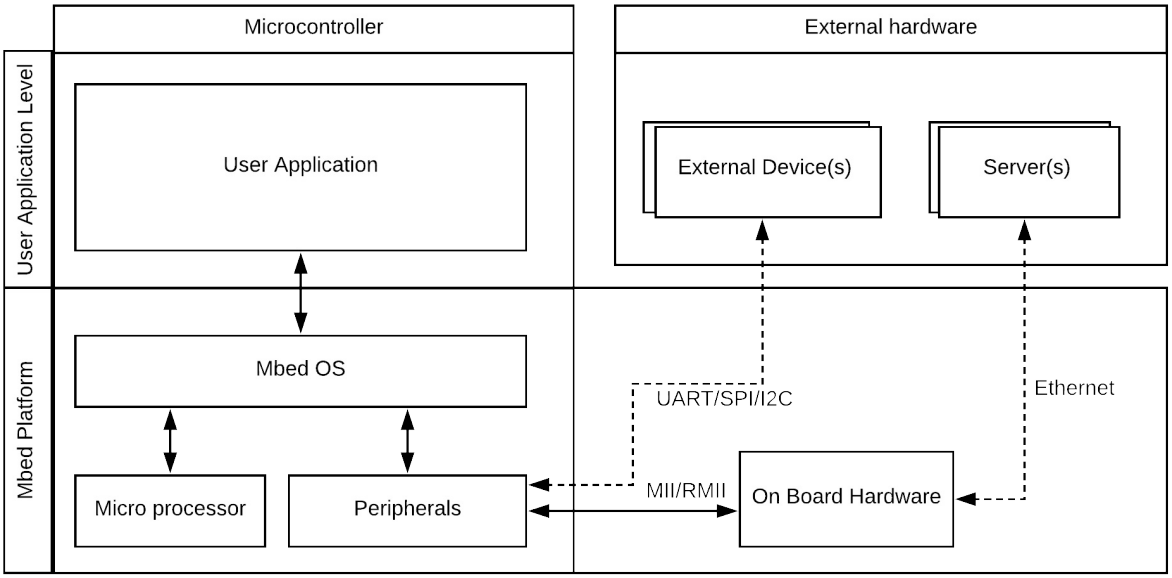


Figure 6. Example deployment scenario.

Hardware Targets

As previously mentioned, Mbed supports countless hardware targets. Below are several vendors list that supply microcontrollers that are supported by MBed OS. Support for more targets keeps growing. Please note that this list only mentions the suppliers and one vendor. For example, ST has multiple series of microcontrollers and each series contains a lot of MCU's. Therefore, providing support

for every microcontroller is not feasible.

- NXP
- ST Microelectronics
- Maxim integrated
- Nordic Semiconductor
- Nuvoton
- Realtek
- Renesas
- Silicon Labs

If a development board of a certain microcontrollers is used as a target, a programmer is usually in the same area to where the board is. This board is connected to a PC with an USB cable. Therefore, to be able to deploy the program to the board, the only additional hardware required is the PC.

Feature Support

With the large number of supported microcontrollers, it is important to note that not all microcontrollers support all of the Mbed OS functionality. Especially Ethernet, LoRa, USB, Bluetooth and Wifi are only supported on a subset of the supported targets.

Evolution Perspective

The first version for ARM Mbed OS was generated on 14th of December in 2014 and it is called Alpha 1, however, only available to Mbed Partners. The second version was released to the public after 2 months called Alpha 2 but it was still heavily in development. Finally, version 3.0 was released officially on 15th of October in 2015. The Mbed OS Release Schedule can be seen in Figure 1, which was shown on ARM TechCon 2014.

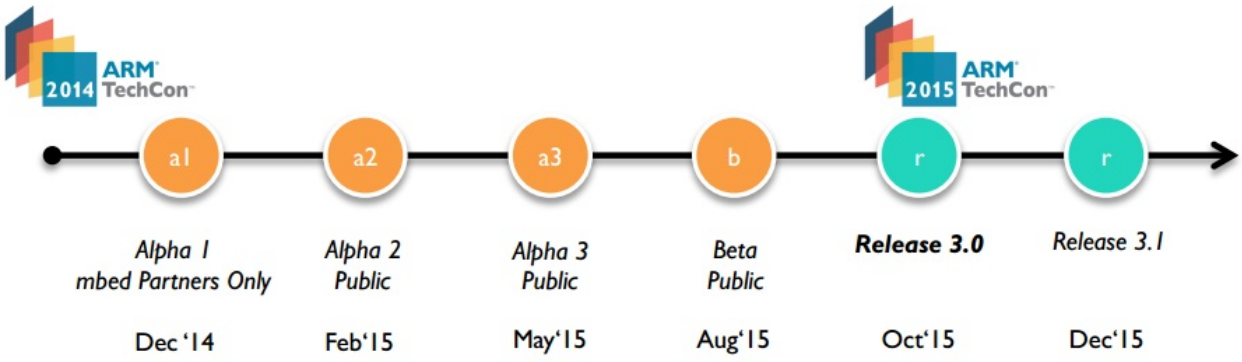


Figure 7. The roadmap of Mbed OS before 2016.

On 5th of August in 2016, Mbed OS 5 with version 5.1 was released. This release has a lot of changes and enhancements so that Mbed OS is usable for many Internet of Things (IoT) use cases. Also, Mbed OS 2 ("Classic") and Mbed OS 3 is merged together such that the ecosystem of Mbed OS 3, such as an RTOS and tooling, is now combined with the ecosystem of Mbed OS 2. Lastly, there are new board included that support Mbed[10]. The process is shown in Figure 2.

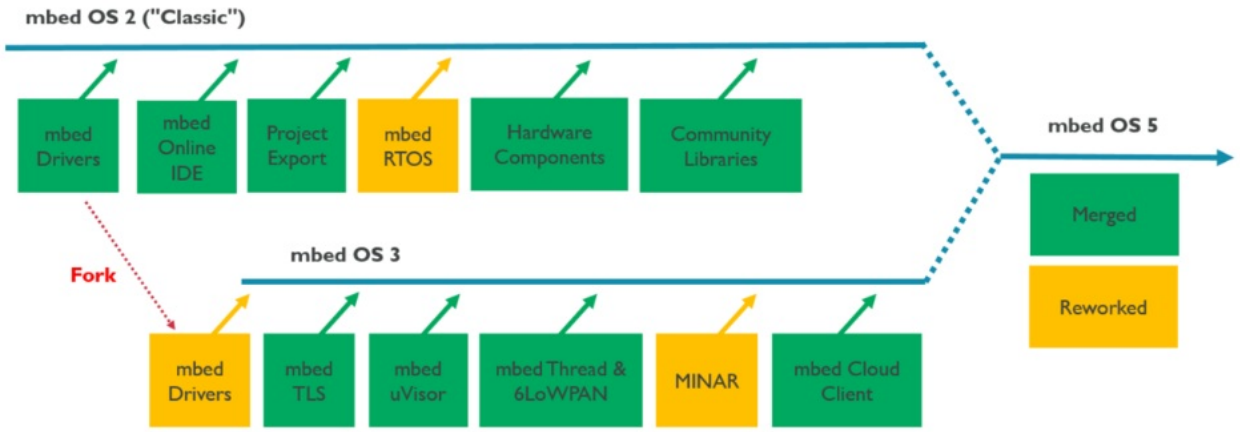


Figure 8. Mbed OS 5 integrates the two codelines of Mbed 2.0 ("Classic") and Mbed 3.0 ("Eventing OS") into one unified platform.

Nowadays, the release process is still continuing and there are three major types of ARM Mbed OS releases which are `major`, `feature` and `patch` [8].

Major releases involve changes to the structure of the OS. These releases are rare since it changes the complete structure of the code. The only major releases so far are version 2, 3 and 5.

Feature releases are created every quarter. These releases add new functionalities to the OS.

Finally, there are the patch releases. These occur every two weeks and include bug fixes, new target boards (or components) and improvements to existing functionalities.

From analyzing the release of Github, it can be seen that they follow these guidelines very closely.

Technical Debt

In this chapter the the technical debt of Mbed OS is discussed. This includes the identification of technical debt, including testing debt through various methods (tools or manual analysis). The chapter is concluded with a section about the evolution of the technical debt of Mbed OS over time.

Identification of Technical Debt

In order to identify technical debt in a large system such as Mbed OS, tools can be used to give an indication of the amount of technical debt. Without these tools, it would cost a lot of time to go through every code. These tools can also reveal locations and tips on how to reduce the technical debt.

Code keywords analysis

Another way to check for technical debt is to look for certain keywords/tags that are commonly used to indicate that something probably needs to be changed. These keywords are also used for communication between developers. However, for identifying technical debt, the following keywords are selected:

-TODO: This keyword indicates that there is an additional feature that needs to be implemented. The analysis revealed that most of the time, these issues relate to either upcoming features or low priority issues.

-FIXME: This keyword indicates that some code is working but it could use some refactoring/rewriting. Also, it can indicate that some code is currently not usable and that it only can be used if it is fixed. Usually, these issues have a greater impact than the TODO issues.

-HACK: This keyword indicates that code is written to circumvent a bug/problem. The issues found in the code which is marked with HACK circumvents certain checks, introduces possibly memory leaks and introduces mismatches between what a function should do and what it actually does.

-XXX: This keyword is more of comment to developers to warn about possible problematic errors or misleading code. These issues can arise because there are no checks or the naming can be confusing depending on the domain the developer is used to.

In order to find the keyword, the following bash command is used:

```
grep -rohinHw "\<keyword\>" --include \\*.h --include \\*.cpp --include \\*.c
```

This command only includes actual source files and it skips documentation files and tools that are written in Python. The reason to exclude these is because they are actually developed in a seperate directory.

The occurrences of each keyword is seperated per module as listed in the table below. Only the modules that have a high occurrence are listed.

Table 3. Overview of occurrences of different keywords in the modules.

Module	TODO	FIXME	HACK	XXX
feature_ble	58	18	0	0
feature_nanostack	153	0	15	6
feature_lwip	171	20	22	10
feature_uvisor	58	5	48	0
feature_storage	41	5	0	0
feature_unsupported	21	20	1	3
targets	255	56	6	0

An example of a TODO is found in the BLE module in the file 'MemorySecurityDB.h':

```
virtual void get_whitelist(WhitelistDbCb_t cb, ::Gap::Whitelist_t *whitelist) {
    /*TODO: fill whitelist*/
    cb(whitelist);
}
```

This clearly shows that there is functionality missing in this function. At the moment, it only uses the callback with the supplied whitelist parameter which is probably an empty list at the moment.

Overall, these keywords indicate there is technical debt that the original author(s) are aware of but could not fix themselves. For instance, an obvious conclusion can be drawn from the table above, the module regarding *lwip* has a large number of keywords `TODO`, `FIXME`, `HACK` and `XXX` in its code. Therefore, when developers want to deploy the module *lwip*, it would not only create more problems involved maintenance but also operates in the way it should not have operated. Thus, the large amount of technical debt in this module could lead to that developers cannot understand this module correctly which leads to a negative impact on the development.

Software Analytics

Below, several interesting analyzed expects are discussed.

Code Duplication

Figure 9. SonarQube code duplication results.

The operating system has a lot of hardware targets. To support each of these targets (efficiently) code duplication is almost unavoidable and is very common for this scenario. The analysis showed most duplication occurs in the C files to support all of the different hardware targets as expected.

Cyclomatic Complexity

Cyclomatic complexity is a metric to measure the complexity of a program. This metric measures the number of independent paths through the code. In the case of MBed, to support all those hardware targets a lot of independent paths exist. This is reflected in the results, Mbed has a very high cyclomatic complexity. The main complexity occurs in the target implementations.

Figure 10. SonarQube detected cyclomatic complexity.

Manual Analysis

In addition to searching for the technical debt using tools, technical debt can be also found manually.

Bug fixes

There are still some issues found in GitHub ARM mbed OS which are labeled as `bug`. This means Mbed OS has some technical debt that need to be paid off and some of these bugs are even dated back to 2014. So far, there are 345 open issues and 98 issues of them are belonging to the type `bug` which takes up 28.4% of all issues. It indicates that Mbed OS team still needs to make much more effort to deal with the technical debt.

Refactoring

In addition to the type `bug`, there is another type called `enhancement` in open issue as well. The goal of this is to improve or refactor some unreasonable code structures, code smell and even API and HAL interface.

Most of them have been reviewed and approved by developers working for ARM Mbed and they improve the performance of Mbed OS significantly, but this also adds to the technical debt. There are 74 open issues about enhancements out of 345 open issues so far.

Testing Debt

Testing debt is caused by the lack of testing or by poor testing quality. In this section, the detail of Mbed OS's testing debt will be analyzed.

Mbed OS uses the service *coveralls* to evaluate the code coverage. The coverage test is triggered once a pull request is submitted. The latest code coverage in Master branch only got 33% which is not a good score.

Only files that are in the *tools* are covered. However, there do exist many other testing files that cover the actual code of Mbed. The reason that these tests are not included is that in the first place, there are no coverage files created for the C/C++ code files thus *coveralls* cannot report statistics about coverage. The reason for this is that most of the C/C++ tests depends on the greentea testing framework which can only run on an actual board. From Mbed webpage about contributing[18] ,it is noted that they test a proposal against the Mbed Enabled development boards. The greentea framework does not have support to calculate coverage percentages but it does look like that most of the functionality is covered by tests.

Evolution of Technical Debt

ARM Mbed OS is in development for many years and has become a popular platform in the Internet of things (IoT) domain. However, the development comes with some historical technical debt which are needed to be paid off. In this section, two types of technical debt are described.

Technical debt of version division

As described in the evolution perspective, Mbed OS needed to keep two version due to radical differences between version 2 and 3. These two versions were split into separate codebases which caused a rift in the developer community to some extent at that time.

For now, the ARM's Mbed team has been working on the combination of Mbed 2.0 and Mbed 3.0 that closes the gap between embedded developers and programmers working at the service layer. This version is known as 5.0 (Mbed 2.0 + Mbed 3.0). This means that the Mbed developers have been struggling to pay off the technical debt that resulted from the version division. The latest released version is Mbed OS 5.7.6 and by looking through the latest version and previous version from the released page, it can be seen that the Mbed team is still concentrating on the compatibility, fixes and changes throughout these years.

Technical debt evolution based on Github repository

For every release for Mbed OS, there are new features added into the new version. In every release, a main developer from ARM Mbed will make an introduction to the new release. This includes a summary of the release in which known issues of critical bugs or changes and which ones are solved. Also, it mentions new port addition the ports addition which are compatible with Mbed OS.

Using the information given in GitHub release[13], two figures have been created. Figure 11 shows the number of ports for upcoming targets. Figure 12 shows the number of fixes and changes that are contributed by contributors in GitHub.

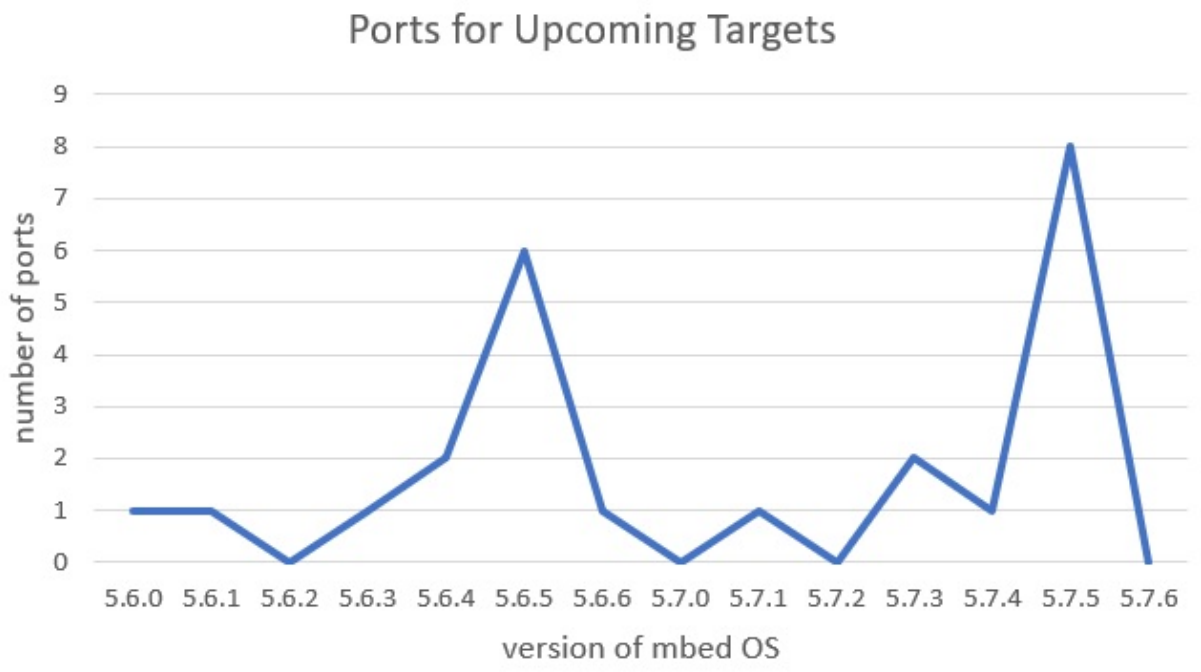


Figure 11. The number of ports for upcoming targets from Mbed OS 5.6.0 to 5.7.6.

Figure 12. The number of fixes and changes from Mbed OS 5.6.0 to 5.7.6

Conclusion

At the beginning of this chapter, the stakeholders were identified that are involved in the development of Mbed OS and the interaction between the system and the stakeholders was analyzed. This can be split into two groups, a developer group which are employed by Mbed, and the community that contribute in different ways such as fixing problems or adding new boards.

Then, in order to understand how the Mbed OS works, the different layers of the source code, common design models as well as the structure of directories was analyzed. It was found that the code was made modular in such way that new boards can be included relatively easy and also that the system contains the most necessary functionalities to support IoT development. Besides, the documentations for each module are comprehensive and provide good descriptions so that users know what each function should do.

Then, the technical debt was analyzed by using tools and doing manual analysis. It was found that although the technical debt has already been decreased significantly, there is still a lot of debt left which is mainly caused by non-compatible boards but also that a lot of tests don't create coverage statistics. This could give the wrong impression of what part of the code is actually tested. There is also a lot of debt that was identified by searching for common keywords such as *TODO*. These revealed that are some implementations that needs to be rewritten in order to support new functionalities or to improve the code quality.

In conclusion, Mbed OS has a good foundation that provides the basic functionalities to create IoT devices. It is also structured such that new boards (with Cortex M processor) can be added and integrated into Mbed OS without changing existing libraries. There is always room for improving the platform by reducing the amount of technical debt but the current debt does not seem to slow down the development since they always hit their intended release schedule. All in all, Mbed OS is a good choice for starting with IoT development with Cortex M processors. This is also reflected by the number of Mbed OS users which is still growing so Mbed OS could become a major player in the IoT sector.

References

1. ARM, Mbed, https://www.mbed.com/en/, Retrieved on 9-3-2018
2. ARM, Mbed Cloud, https://cloud.mbed.com/, Retrieved on 9-3-2018
3. ARM, Mbed OS, https://www.mbed.com/en/platform/mbed-os/, Retrieved on 9-3-2018
4. Rozanski, N. Woods, E. , 2012, Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley.

5. Mbed OS Style guide, https://docs.mbed.com/docs/mbed-os-handbook/en/latest/cont/code_style/, Retrieved on 23-2-2018
6. Testing in mbed OS 5, https://docs.mbed.com/docs/mbed-os-handbook/en/5.2/advanced/testing/, Retrieved on 24-2-2018
7. ARM, mbed OS Software Design Guide, https://docs.mbed.com/docs/mbed-os-handbook/en/5.2/cont/design_guidelines/, Retrieved on 8-3-2018
8. ARM, How We Release Arm Mbed OS, https://os.mbed.com/docs/v5.7/introduction/how-we-release-arm-mbed-os.html, Retrieved on 4-3-2018
9. ARM et al, Github: ARMmbed mbed-os, https://github.com/ARMmbed/mbed-os, Retrieved on 2-3-2018
10. ARM, Introducing mbed OS 5, https://os.mbed.com/blog/entry/Introducing-mbed-OS-5/, Retrieved on 15-3-2018
11. ARM et al, Github: PR #6102, https://github.com/ARMmbed/mbed-os/pull/6102, Retrieved on 3-4-2018
12. ARM, mbed OS forum, https://os.mbed.com/forum/mbed/topic/28559/?page=1#comment-54245, Retrieved on 16-3-2018
13. ARM et al, Github: Release page, https://github.com/ARMmbed/mbed-os/releases, Retrieved on 16-3-2018
14. SonarSource, SonarQube, https://www.sonarqube.org/, Retrieved on 16-3-2018
15. SonarOpenCommunity et al, Github: SonarQube C++ plugin (Community), https://github.com/SonarOpenCommunity/sonar-cxx, Retrieved on 15-3-2018
16. ARM, Mbed OS code repository, https://os.mbed.com/code/, Retrieved on 25-3-2018
17. ARM, Mbed OS handbook, https://os.mbed.com/handbook/Founders-interview, Retrieved on 22-3-2018
18. ARM, Mbed OS contributing, https://os.mbed.com/contributing/, Retrieved on 4-4-2018
19. ARM et al, Github: Pull request number 6341, https://github.com/ARMmbed/mbed-os/pull/6341, Retrieved on 4-4-2018
20. ARM et al, Amazon CI build server: build number 5581 for target board FF_LPC546XX, http://mbed-os.s3-eu-west-1.amazonaws.com/builds/5581/PASS/FF_LPC546XX/GCC_ARM/5dd46136ada7e3f03833fd8255ba6ffac85004e1_build_log_FF_LPC546XX_GCC_ARM.txt, Retrieved on 4-4-2018

osu! - a fast-paced, community-driven rhythm game for PC

From top left, top right, bottom left, bottom right: Vincent Bejach, Jonathan Levy, Maiko Goudriaan, Pavel Rapoport

Abstract

Osu! is an open-source free-to-play community-driven rhythm game, aiming to centralize different game modes from arcade or console competitors. It is built in C#. The project seems well structured, and present a manageable amount of technical debt, although the documentation has been found to be lacking. A significant issue however would be social debt: very little information on the design strategy or on what is needed is communicated to the developers, which could make contributing to the project harder.

Table of content

Introduction

Rhythm games have historically been mostly developed on arcade or gaming console. However, since 2007, *osu!* offers a pc alternative, gathering game features from many arcade and console competitors. On August 2016, a new major version of *osu!* called *osu!lazer* has been made available as open-source software. The goal was to make *osu!* available on more platform and improve transparency.

The aim of this chapter is to analyze the architecture of this project and to present an overview of the system. To analyze it, we will refer to the conceptual framework provided by Rozanski and Woods. We will first cover *osu!*'s features and stakeholders, then we will discuss the architecture itself through context, functional and development views. Then, we will discuss *osu!* from a usability perspective. Indeed, as a rhythm game, *osu!*'s interface and design have a huge impact on the player's performance. Finally, the technical and social debt of the project will be analyzed.

Stakeholder analysis

Let's describe the stakeholders of our project.

As a disclaimer, the information has been verified on 22-02-2018 and will be subject to changes. This stakeholders analysis is providing a snapshot of the state of osu!.

Summary of the stakeholders

We can summarize most of the stackholders in following picture:

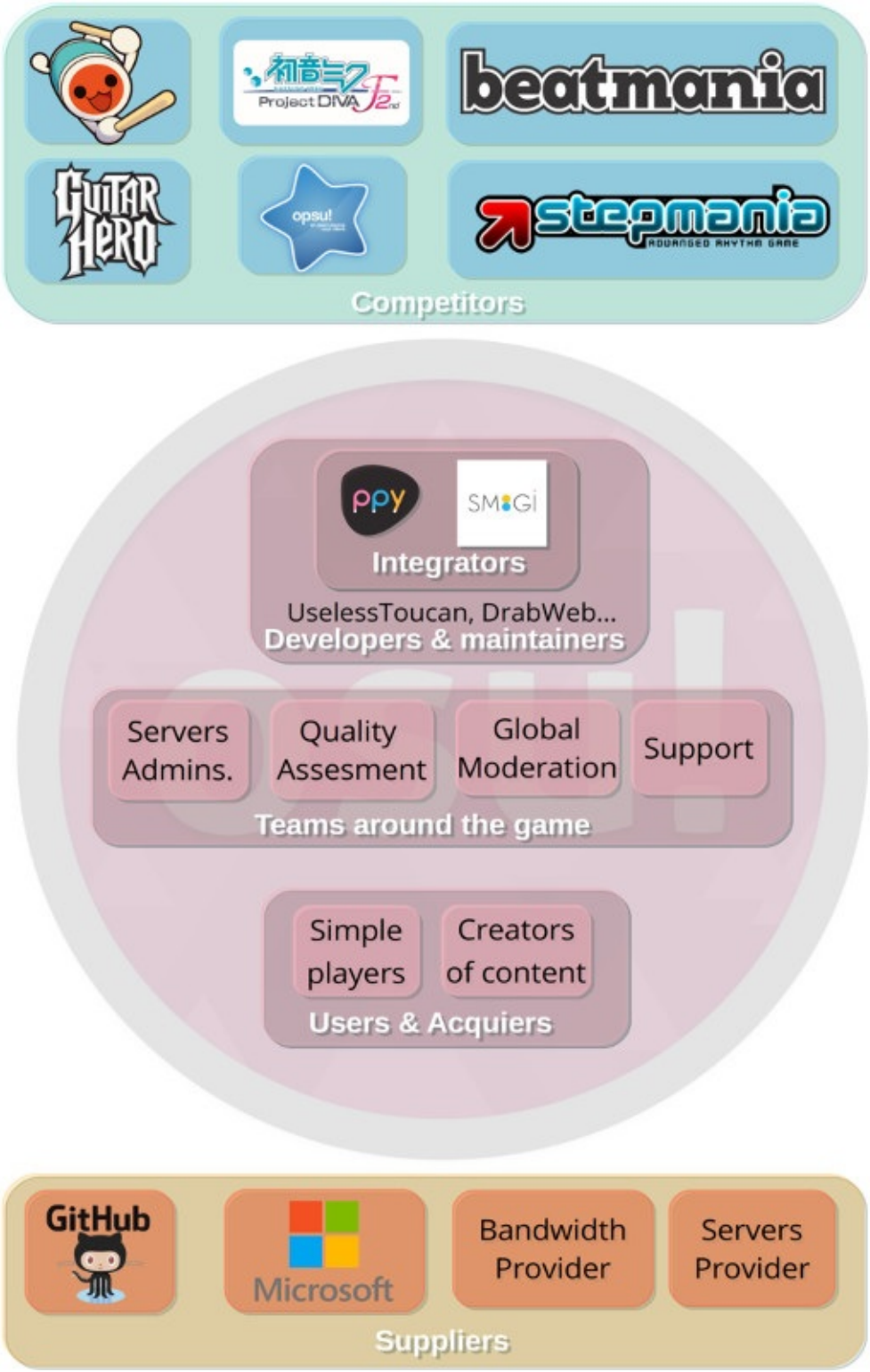


Figure 1 - *Stakeholders for osu!*

We listed them with the following categories.

Stakeholder	Description
Users	Passive (playing), active (creating content)
Acquirers	same as users
Developers, maintainers	*peppy*, *smoogipoo*, community
Testers	Developers + subset of users
Adminitrators	users (running the game), *nekodex* and *nanaya* (website)
Assessors	Quality Assurance Team, Global Moderation Team, Language Moderators
Communicators	*peppy*, various forum members
Sponsors	osu!supporters (through merch store)
Suppliers	GitHub, Microsoft, ISP, servers owners
Competitors	Beatmania, Stepmania, Guitar Hero ...

Some of the stakeholders need extra attention, we will review them specifically.

Users

osu! players can be active at various degrees. They can:

- enjoy the game
- create new beatmaps
- create new skins
- help people on the forum
- report issues

Users only enjoying the game are called "passive users", while content creators are called "active users".

Considering all of this, the users appear in several categories of stakeholders.

Assessors

The assessors are mainly composed of three teams of volunteers:

- Quality Assurance Team (QAT): they check that the new content created by users is of good quality.
- Moderation Team: they operate on a communication level to ensure rules are followed on the in-game chat, the forum, the comments on the website...
- Language Moderators: they are language-specific moderators for subforums, because of the worldwide community.

Sponsors

To financially support the game, users can buy a supporter's badge on the website, allowing some convenient (but non-crucial) features, like in-game content downloading. Additionally, a merchandise store lets you buy several goodies to help supporting the development. No precise information about the goodies manufacturers could be found.

Developers

Some developers come and go, others proved to be central in the development process. The following informations about developers are acquired from the github repository of the project. osu! was open-sourced in September 2016, so it's difficult to get any detailed information about contributors before this date. As far as we know, *peppy* was mostly the only one working on the project before it was open-sourced.

Contributor	Commits	LOC++	LOC--	Active during
peppy	3039	153480	117840	09.2016 - present time
smoogipoo	1719	69818	45740	01.2017 - present time
DrabWeb	467	29023	15659	01.2017 - 09.2017
EVAST9919	344	10920	6422	12.2016 - 11.2017
huoyaoyuan	319	8109	5847	09.2016 - 10.2017
Tom94	239	6334	5755	09.2016 - 09.2017
SirCmpwn	220	13117	6898	09.2016 - 03.2017
UselessToucan	45	3954	2896	10.2017 - present time

As this table shows, *peppy* is concentrating more than half of the commits, and *smoogipoo* a fourth. With the contribution history, we conclude that *peppy* (Dean Herbert, founder of osu!) and *smoogipoo* (Dan Balasescu) are the only persons who continuously developed this project from the moment it was open-sourced (or for a very long time). Other developers keep coming and going. Current developer team roster is located on the osu! website.

We can summarize the contribution of previously mentioned contributors with a timeline.

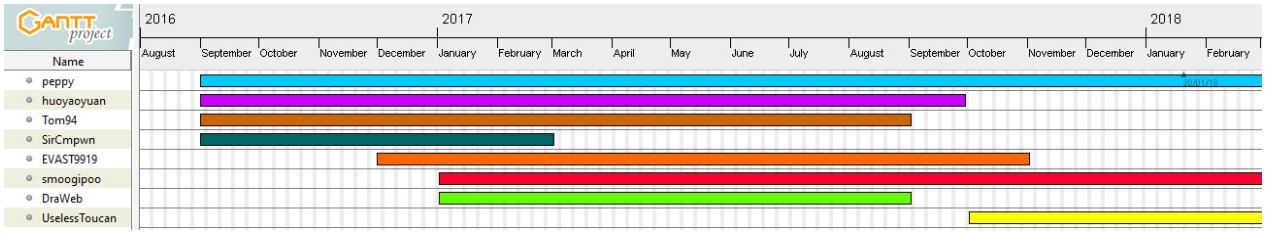


Figure 2 - *Timeline describing developers implication*

To order visualize the stakeholders, we present them in a Power-Interest Grid. Users can be quite different, so they are represented as an area.

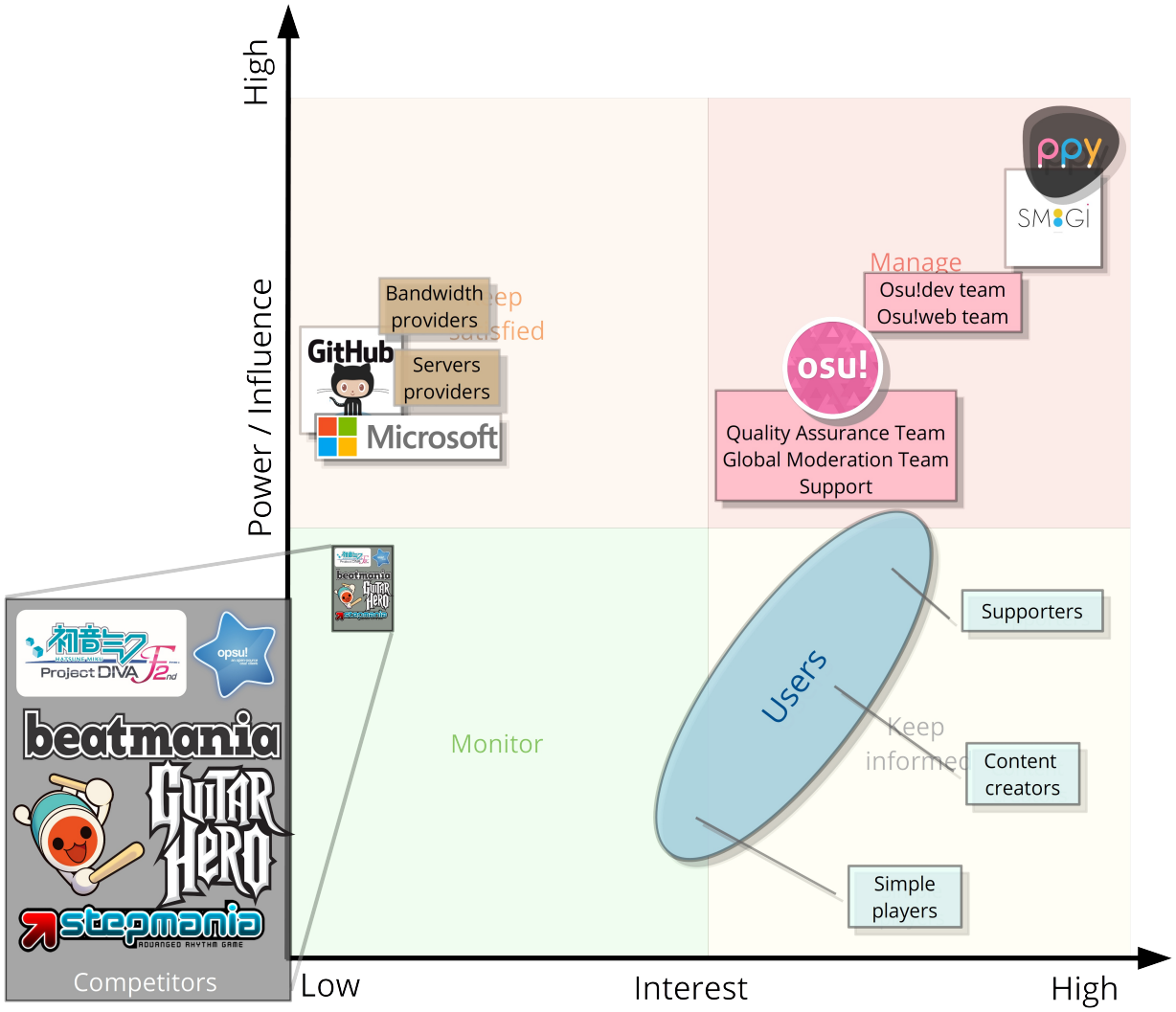


Figure 3 - *Power-Interest grid for stakeholders.*

Context view

The context view describes the relationships, dependencies, and interactions between the system and its environment.

System scope and responsibilities

Osu! is the *"bestest free-to-win rhythm game"* that provides entertainment to over 11 million users worldwide, who have so far played over 7.62 billion ranked games. *Osu!* can either be played unranked or in a competitive ranking system.

After *osu!* was open-sourced, it became possible for a user to contribute to *osu!* via code. Besides that, a user can also create and share their own playable content. It can be via beatmaps or via skins applied to game elements.

External entities

In this section we show and briefly explain the external entities related to *osu!*. We first provide an overview of these external entities.

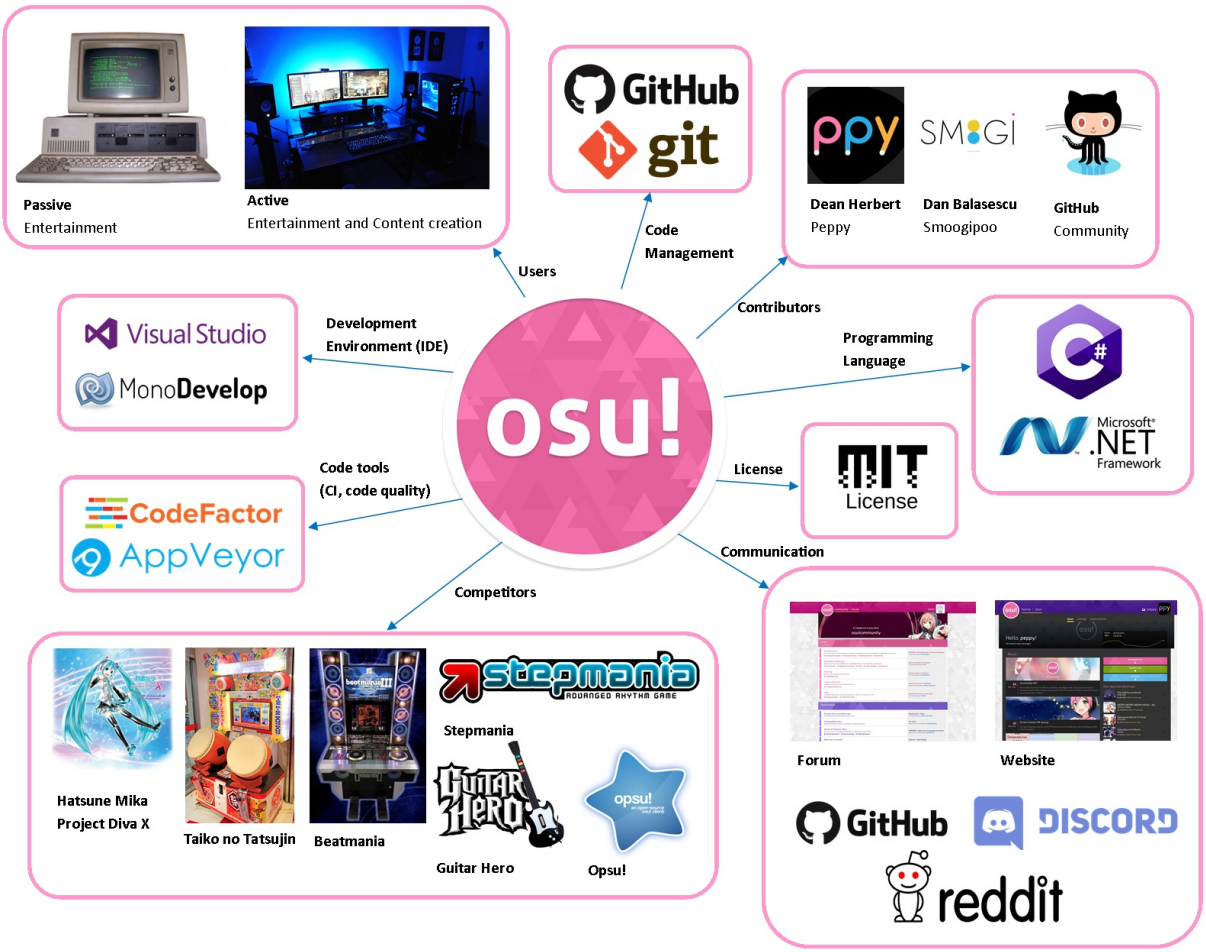


Figure 4 - *Entity view for osu!*

Technical aspects

Osu! is written in C# which is part of the .NET framework created by Microsoft. It is mainly developed in Visual Studio (Windows and macOS) or MonoDevelop (Linux). Version control and code management are done via Git, using a public GitHub repository. For continuous integration AppVeyor is used, with CodeFactor for automatic code review. The code is developed under the MIT License.

Developers

The core developers of *osu!* are Dean Herbert (peppy) and Dan Balasescu (Smoogipoo). With the help of other developers from the GitHub community, *osu!* is being maintained and extended.

Community

The user of *osu!* are separated in two groups:

- Passive users
- Active users

The groups are categorized by the same attributes as explained in the stakeholders section, thus passive users mainly play *osu!* for entertainment and active users also actively contribute to it.

The communication between developers goes through GitHub repository or Discord. The communication between users and developers happens through the website and forum. The communication between the users happens mostly through the forum and Reddit.

Competitors

The same competitors as in the stakeholder section are used. As a reminder, a large part of the competitors only exist in an arcade format, or on game consoles. There are only a few competitors playing on the same platform, namely Stepmania, Guitar Hero and Opsu!.

External Interfaces

Here are the external interfaces used by *osu!*. Interfaces are external sources of code which contain functionalities that can be used in *osu!*. This saves the developers the effort of creating (complicated) code. First, the overview is illustrated then the different interfaces are explained.

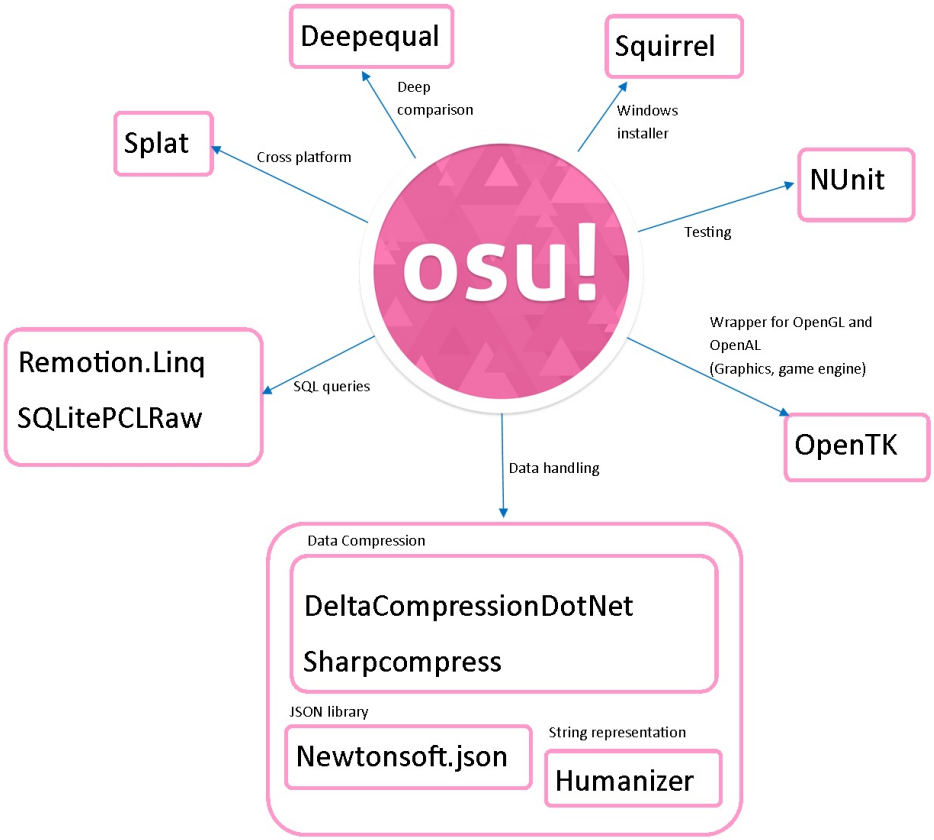


Figure 5 - *Interfaces around osu!*

The following table describes the interfaces used by *osu!*:

Library	Functionality
Deepequal	Extensible deep equality comparison library
DeltaCompressionDotNet	Wrapper around Microsoft's delta compression application programming interfaces
DotNetZip	Manipulating zip files
Humanizer	Manipulating and displaying strings, enums, dates, times, timespans, numbers and quantities
Newtonsoft.json	JSON framework that can easily transform objects into JSON and vice versa
NUnit	Most popular unit test framework for .NET
OpenTK	Wrapper for OpenGL and OpenAL, libraries for creating graphic and sound interface (game engine)
Remotion.Linq	Parsing LINQ expression trees and generating queries in SQL or other languages
Sharpcompress	Library for dealing with many different compression formats
Splat	Used for cross-platform manipulations
SQLitePCLRaw	Portable Class Library for low-level access to SQLite
Squirrel	Toolset for managing installation and update of software on a Windows platform

Functional view

According to Rozansky and Woods, functional view "defines the architectural elements that deliver the function of the system being described". This means its purpose is to demonstate how *osu!* perform these functions.

Each functional element is responsible for a certain feature. Responsibilities of functional elements may be fine-grained (for example, "position target circle at specific coordinates") or coarse-grained (such as "creating new beatmap"). We would prefer to stay on a higher level of abstraction to keep our model simple.

We start our analysis by listing the most important features and aligning them to a functional element, responsible for performing this feature.

Functional elements	Features
User information system	• Log-in • Ranking • Account information
In-game language selector	• Internationalization
Beatmap editor	• Creating or editing beatmaps
Gameplay core	• Standard mode • Taiko mode • Catch mode • Mania mode
Multiplayer	• Head to Head mode • Tag Coop mode • Team Vs mode • Tag Team Vs mode
Communication	• In-game chat
In-game Tweak Tool	• Game settings modification

So from features, which are obviously visible to users, we moved to functional elements. Now it's time to show how they are connected.

In the center of diagram we position the **Gameplay core** element. This function depends on the **In-game language selector** and the **In-game Tweak Tool**. Both of this tools could change visualisation, also the **In-game Tweak tool** can change gameplay options. Optionally it depends on the **Beatmap editor** as you can either create your own beatmap or just download an existing one. Also, there are mutual dependencies with the **User information system**. Depending on the user status, *osu!* client can, for example, enable automatic downloads of beatmaps for multiplayer. On the other hand, the **User information system** needs information from the **Gameplay core** to provide proper ranking.

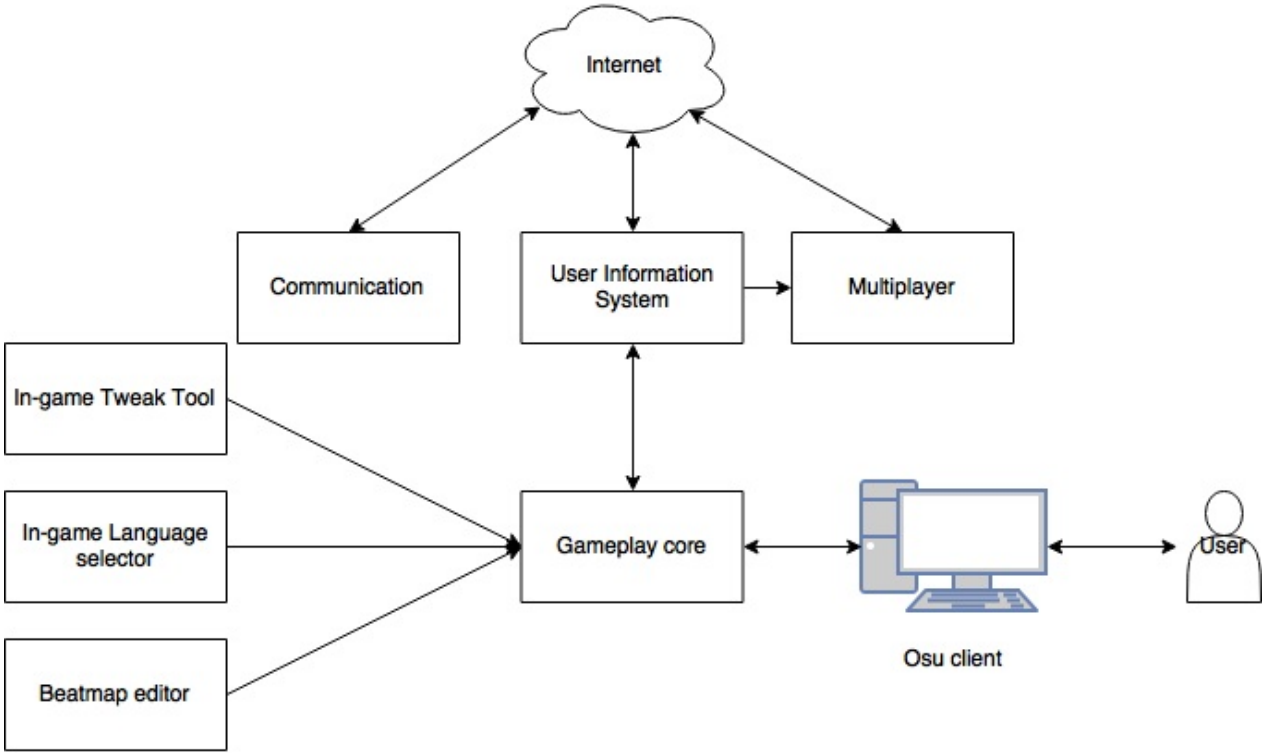


Figure 6 - *Overview of the features*

As we said, the **Gameplay core** is responsible for running the game. This is quite obvious without complex analysis. So we would like to unfold the **Gameplay core** one more level.

First we jump to the Game project. There are a lot of most derived classes that contain little functionality. However, it contains a calculation logic for different purposes: result calculation, for example, used in processing rankings, hit point calculation which depends on the user settings and game mode etc. This logic is used by Ruleset.X projects, where X stands for a game mode. These projects contain rulesets for the specific game modes. Also the Game project is responsible for arranging menu elements and different screens of the application, while Ruleset.X projects are responsible also for arranging gameplay screen for a given mode.

The Resources project, as its name suggests, contains various resources used by the application, such as fonts, music, sounds, textures etc.

Osu.Game and Osu.Game.Rulesets.X projects rely heavily on the Framework project which is the framework, developed specifically for *osu!*. This project is even put in separate repository on GitHub. So it would be interesting to make a step into Framework, as it is an important interface for *osu!*.

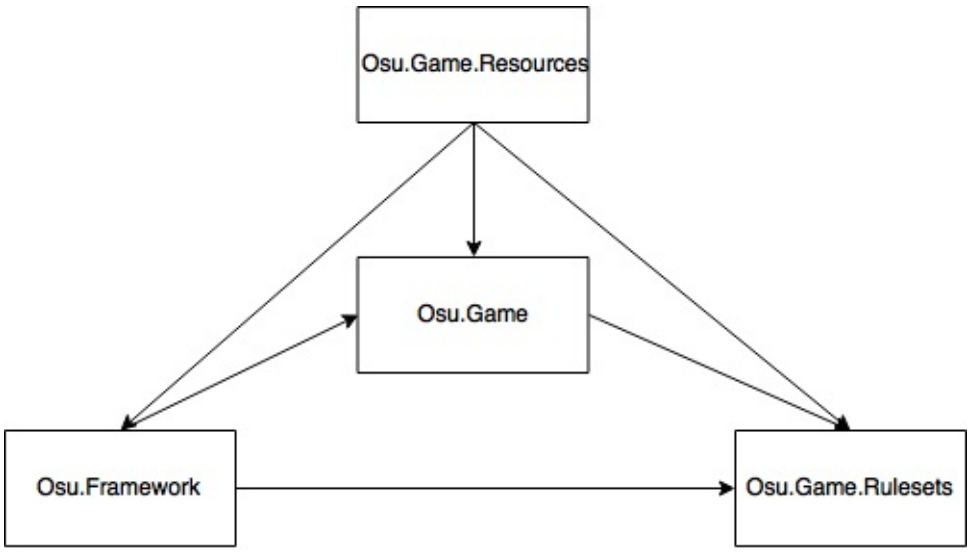


Figure 7 - *Detailed view of the modules in the main project*

In the Framework we find a huge tool library. All modules are sorted by their purpose. Detailed explanation of all mechanisms provided by this framework would take a lot of time. We will point out the most important ones, such as threading system, platform interaction and input processing.

Firstly, Threading section contains different thread types and a Scheduler class, which is the heart of the threading system. Apparently, *osu!* requires more fine-grained threading management. Standard C# tools don't offer the software tools required to keep a high frame rate when dozen of blinking targets appear on screen. Also, it is more convinient to have different thread types for different purposes. Therefore the special threading system was created with different type of threads and the scheduler, which is responsible for managing threads.

Next, Platform section is dedicated to interaction with the Operating System (OS) on which *osu!* is running. It would be nice to be able to use the clipboard between in-game chat and the OS, or to be able to connect to the server in order to show your skills and climb up the wordwide leaderboards. Platform section provide tools for *osu!* to interact with the OS and use some of its functionality.

Finally, if the user would want to interact with the game we need Input processing section. It uses OpenTK library to gather raw input from keyboard and mouse. After that it organizes signals in a convenient manner for further processing. Also, this module binds key combinations to a certain game functions.

Also we should just mention other modules: Allocation, which is responsible for memory management; Audio, for processing and editing in-game music; Graphics, which contains everything that is needed to create in-game visual components and configure their behavior.

So, now we can answer the following correlated questions about these projects functionality "What does THIS thing actually do?" and "How is THIS thing working?". *osu!* menu interface and game modes, implementing features of this game, are located in separate projects. This projects extensively use Framework tools for internal logic and visualization.

Development View

Now, we will cover the code structure and development process of *osu!* to understand what architecture and what methods of standardization are used in it.

Module Organization

The project is distributed across three repositories:

- osu-framework, acting as a top layer processing I/O, with which the user interacts to play
- osu-resources, containing the assets used by the application
- osu, the principal game module, containing the game logic

The distinction between `osu` and `framework` was made to allow Separation of Concerns, and focus `osu` on game logic rather than on I/O processing.

The architecture of the modules and their mutual dependencies are shown below:

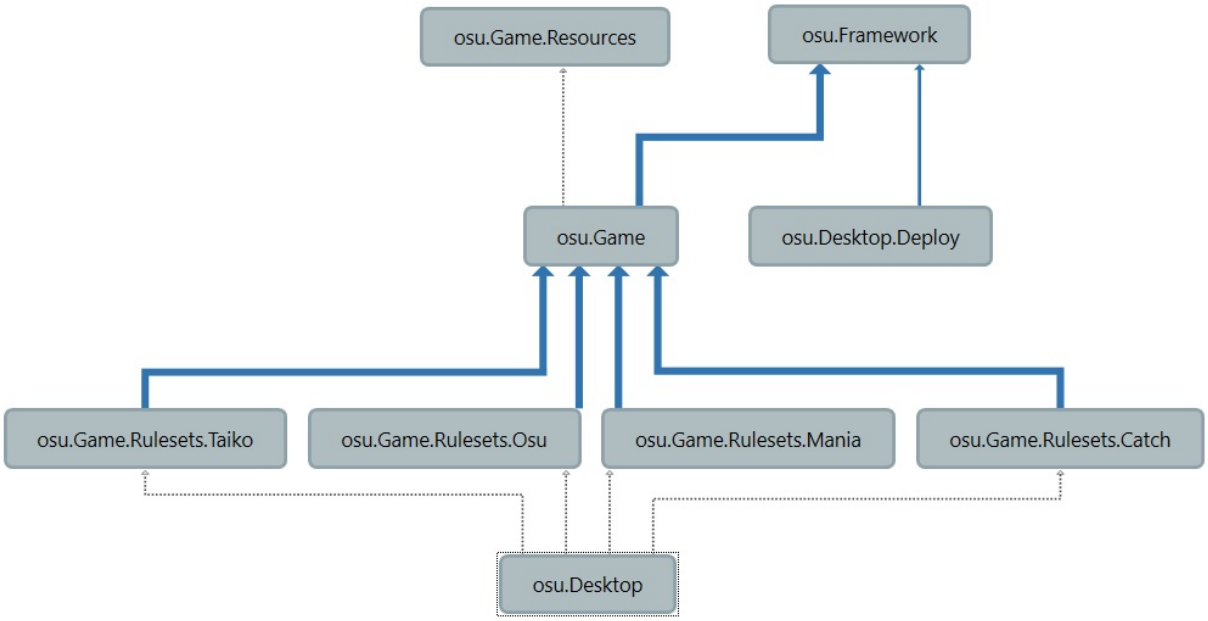


Figure 8 - *Modules architecture (with transitive arrows)*

Blue arrows represent class calls, while grey ones are project references. Please note that transitive calls are simplified here, and that lower game modules (Taiko, Osu, Mania, Catch) also call the framework directly. Full dependencies are shown below:

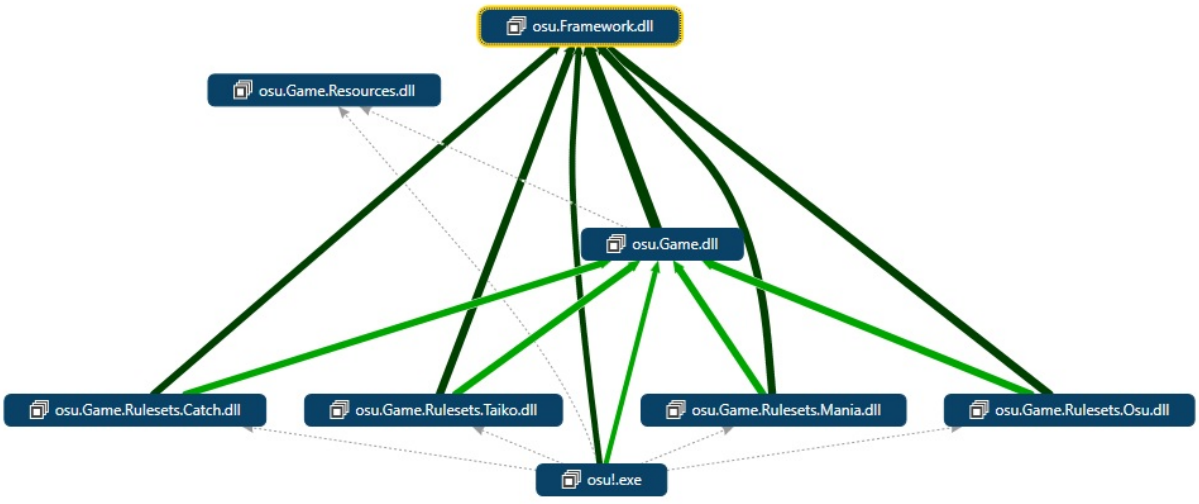


Figure 8 - *Modules architecture (with full dependencies)*

This structure is very concentric, with every module calling the framework. This was expected, as the framework was designed to centralize I/O and all modules use them. Together with Game, they act as "core" component used by game modes as "customizations".

Codeline Organization

Code storage structure

Codeline is organized in the following structure:

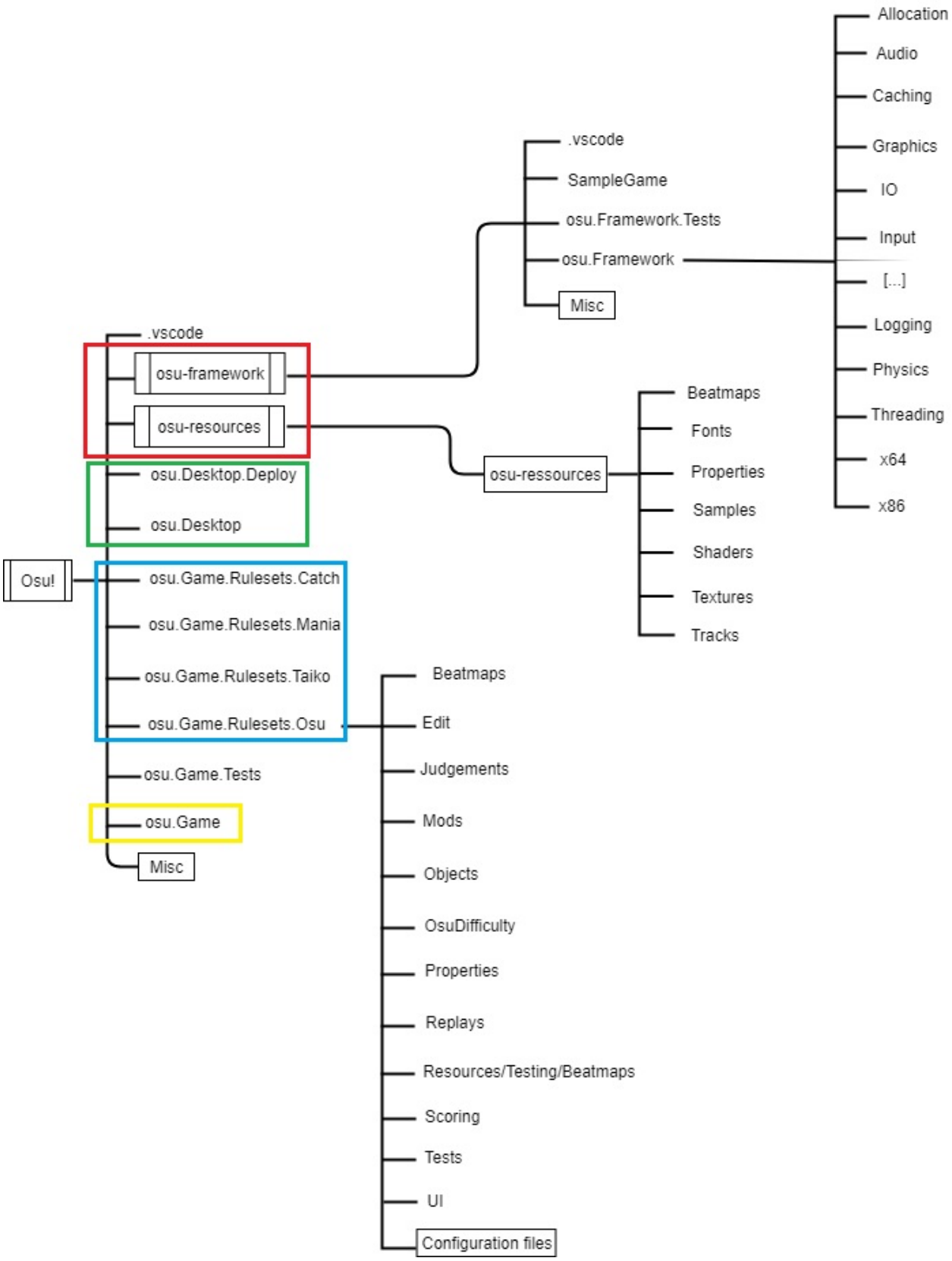


Figure 9 - *Codeline organization*

The root directory also contains both osu-framework and osu-resources modules. The following table summarizes the different groups of folders:

Group	Color	Description
Support code	Red	Code and files on which the game code relies
Application code	Green	Code and files used by desktop application on which the game is played (debug & release)
Rulesets	Blue	Code defining the different game rules and objectives of *osu!*'s four game mode
Game	Yellow	Source code of the game itself
Miscellaneous	/	Rest of the files not mentioned (among which configuration & license files)

As seen before, Framework is big, and the resources section contains the assets used by the game. The Ruleset folders all contain similar files: for instance files defining the gamemode beatmaps specificities, the judgments (game won or not, etc).

Although designed separately from Game, Framework would hardly be usable for anything else, since its components seem to be built for dealing with *osu!*-specific components.

The codebase is managed using the GitHub workflow, as well as by some guidelines strictly enforced by the integrators.

Building and release method

Osu! uses continuous integration tool Appveyor to build its successive versions. This allows to get immediate info on the state of the code at every merge. When finished, the application will be released as standard game installer, including an online updater. For now, there are regular GitHub milestones, to help organize the development team.

Common Design Models

We noticed some common processing methods:

- The logging is done consistently throughout the project, using a class from Framework.
- Framework itself, as it is called everywhere, constitutes a standard processing method for I/O.

Standardization of design

State Pattern

This state pattern is used to handle the input of the keyboard and mouse. First, the current state of the input device is captured, then it is further processed.

Adapter Pattern

There are many beatmaps in the *osu!* community and multiple game modes. To overcome the problem that a dedicated beatmap has to be created for every song, gamemode and difficulty combination, the application uses the adapter pattern. It uses the `BeatmapConverter<T>` as base class to convert a beatmap. Below is a part of the adapter class for the osu!catch game mode.

```
protected override IEnumerable<CatchHitObject> ConvertHitObject(HitObject obj, Beatmap beatmap)
{
    ...
    ...
}
```

Factory and Method Factory Patterns

We noticed several instances of factory classes in the codebase. Among those, we noted a few private factory inner classes in the project and four factory methods in `CustomizableTextContainer`. The following code snippet shows one of them.

```
/// <summary>
/// Adds the given factory method as a placeholder. It will be used to create a drawable each time [<paramref name="n
ame"/>] is encountered in the text.
/// The <paramref name="factory"/> method must return a <see cref="Drawable"/> and may contain an arbitrary number of
 integer parameters.
/// If there are, fe, 2 integer parameters on the factory method,
/// the placeholder in the text would need to look like [<paramref name="name"/>(42, 1337)] supplying the values 42 a
nd 1337 to the method as arguments.
/// </summary>
/// <param name="name">The name of the placeholder that the factory should create drawables for.</param>
/// <param name="factory">The factory method creating drawables.</param>
protected void AddIconFactory(string name, Delegate factory) => iconFactories.Add(name, factory);
```

Standardization of testing

Several tests modules are created with VisualTest, a Visual Studio feature. It allows to create dedicated views to test classes or modules in separate environments. The tests are validated by inspection and semi-automatic execution: a user still has to initiate the testing, it is not part of CI flow.

The project also has around 150 unit tests, checked by Appveyor. This is unusually few, but we'll discuss it in the technical debt section.

Usability Perspective

Usability is a major concern with software, even more in a competitive environment: if users don't interact naturally with your software, they won't use it unless forced to. *Osu!*'s usability is all the more important considering the fact that a rhythm games player heavily relies on their muscular and visual reflexes when playing. Consequently, a clear, readable interface with adapted peripherals is very important.

For this part, we will sometimes compare the interfaces of the current software and its legacy version. We will call the current game that we have analyzed so far "osu!lazer", while the old version will be called "legacy osu!".

Touch points

First, let's review the touch points of osu!lazer, ie all the places where the user interacts with osu!lazer. The following graph shows how the user jumps from one touch point to another. The peripheral used for each is mentioned. The term "pointing device" is grouping both mouse and drawing tablet, as many players are using a tablet to be able to aim the targets using absolute positioning. Keyboard-oriented interactions are in blue, while pointing-oriented interactions are in pink. Logically, touch points relying on both peripherals are in violet.

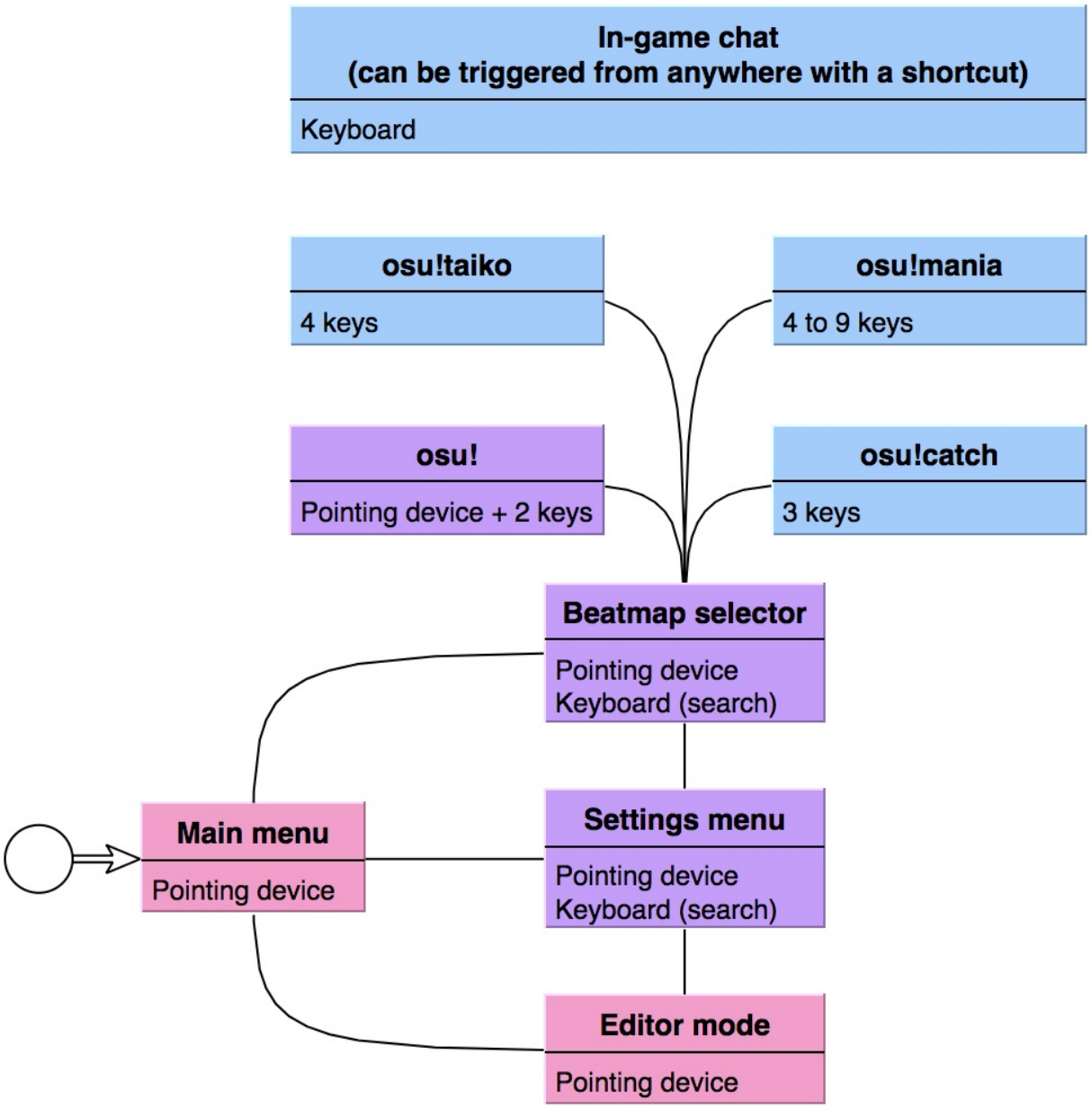


Figure 10 - *Touch points in osu!'s interface*

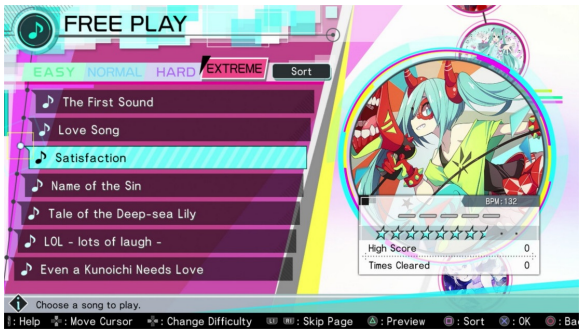
This graph shows that osu! (both osu!lazer and legacy osu!) can be played with usual PC peripherals.

Interface

"A user interface is like a joke. If you have to explain it, it’s not good." Following that motto, osu!lazer's interface is straightforward, especially considering how close to other rhythm games interfaces it looks like.

From left to right, top to bottom, here are the beatmap selection screens for each of the following games: Hatsune Miku Project Diva, Beatmania, Guitar Hero, and the current in-development osu!lazer.

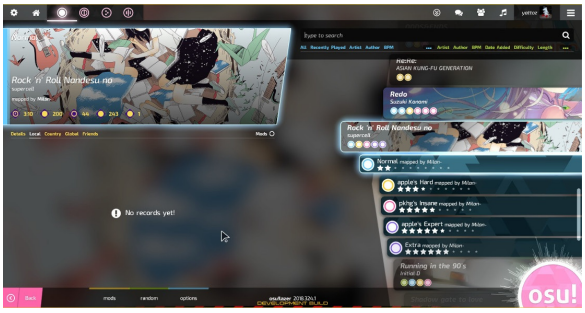
Beatmania being an arcade game, the picture comes from a photograph. We couldn't find any better pictures and apologize for this.



Hatsune Miku Project Diva X (Console)

Guitar Hero Live (Consoles)

Beatmania IIDX (Arcade)



osu!lazer (PC)

We can notice several similarities.

- All games have a list of beatmaps (although the term "beatmap" is osu!-specific, the concept is the same).
- They are arranged into a vertical list.
- The difficulty is rated with a number. Most games are using a star-counting system. Beatmania is an exception which only uses numbers.
- Some art about the current highlighted beatmap is shown on the side.
- Also on the side, the current high scores are displayed.
- All games have a sorting function. Still, this feature is more developed for osu! for two major reasons:
  - First, you can download as many beatmaps as you want, unlike the other games who have a limited number of beatmaps,
  - Being on PC, you can use the keyboard to search, which is impossible on other platforms.

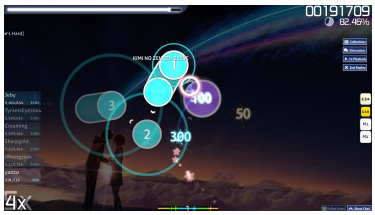
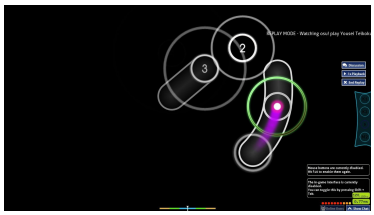
All these common points allow players coming from any rhythm game to quickly adapt to osu!lazer's interface.

Adaptability to the user

With millions of players all around the world, *osu!* is trying to satisfy a large spectrum of different users that have their own habits or feelings. To do so, it presents several levels of variability.

The first one, as mentioned above, is the pointing device. The most obvious interaction devices to play osu! are the keyboard and mouse. But both legacy osu! and osu!lazer are compatible with drawing tablets, and experienced players tend to prefer them because of the absolute pointing possibility. Technically speaking, this has been realized by implementing raw input instead of relying on the OS mouse input. Another advantage of raw input is to bypass any OS artificial acceleration to make the cursor move the same way your peripheral moves, enhancing accuracy and natural use.

The second level of adaptability is the skin engine. Legacy osu!'s appearance can be completely changed by downloading and installing a skin. Although this feature has been implemented on osu!lazer to some extent, its support is not complete yet. Again, most experienced players recommend using a skin to clarify the game modes, improve readability and thus reduce reaction time. As an example, this is how osu!lazer and legacy osu! appear without skin, and legacy osu! with a custom skin.

		
osu!lazer with default skin	Legacy osu! with default skin	Legacy osu! with custom skin

A dim and simple skin as shown on the right is a huge help for cleaning up the interface and leaving only useful visual cues for the gameplay. For example, removing the background video, the different colours of the targets, and having a thicker white border helps reading the beatmap.

Apart from these two major variability points in terms of usability, both osu!lazer and legacy osu! present numerous parameters to tailor your gaming experience. For instance, you can disable the parallax settings (that can make some people dizzy), or you can adapt the frame lag to match the response time of your screen. These settings are available to be sure the player can experience the game in the best conditions. Usability has been thoroughly studied to provide both quick adaptation and high configurability.

Technical debt

Technical debt represents the amount of additional work required in order to improve the code quality. In order to measure that of the *osu!* application we used four tools and conducted manual analysis. The first tool is CodeFactor, which is also used by the developers themselves, by being integrated in the repository. The additional tools are SonarQube, Visual Studio metrics (VS metrics) and ReSharper.

Evolution of Technical debt

In the next figure we illustrate the evolution of technical debt.

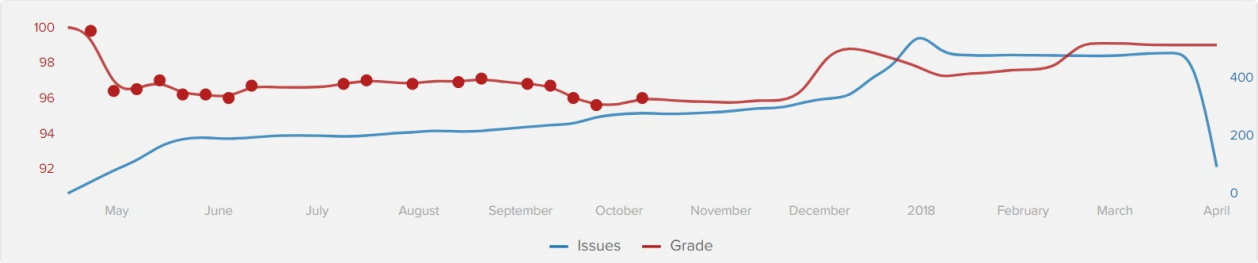


Figure 11 - *Evolution of technical debt through time*

The first drop can be explained by code being added to the project. After that the technical debt stays constant, however manual inspection indicated that new files barely introduce any debt while modifications tend to add some. In October 2017 a label concerning technical debt was introduced on GitHub. From that moment, improvements were made paying technical debt (noticeable in December). Furthermore, recent modifications added little debt.

Tool analysis

Overall score

The overall scores from the different tools are good, with an average grade A from SonarQube (concerning technical debt) and CodeFacor. VS metrics gives 72/100 points for code maintainability which is a decent score, however this metric is a rather obsolete. Therefore, we decided to overlook its value and looked at the issues found and used manual inspection to verify. Unfortunately, ReSharper does not provide a total score.

In the next figure the distribution of the technical debt calculated by SonarQube is displayed, highlighting a few outliers containing high amount of technical debt, although their grade is still reasonable.

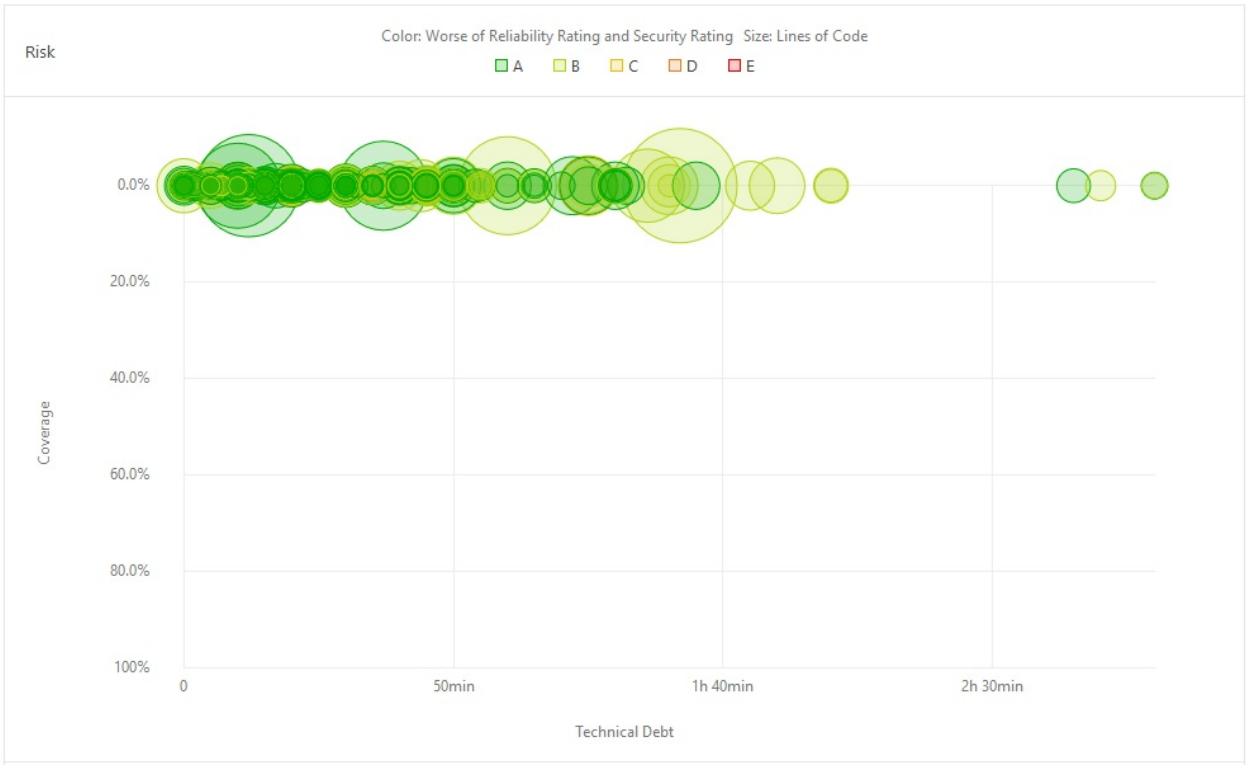


Figure 12 - *View of technical debt by files, in estimated number of hours needed to fix it*

SonarQube gives two additional scores for bugs (C) and vulnerabilities (B). It is worth noting that a large part of the grade for bugs is due to the lack of asserts in test cases. The next figure contains part of the dashboard of SonarQube including the different ratings.

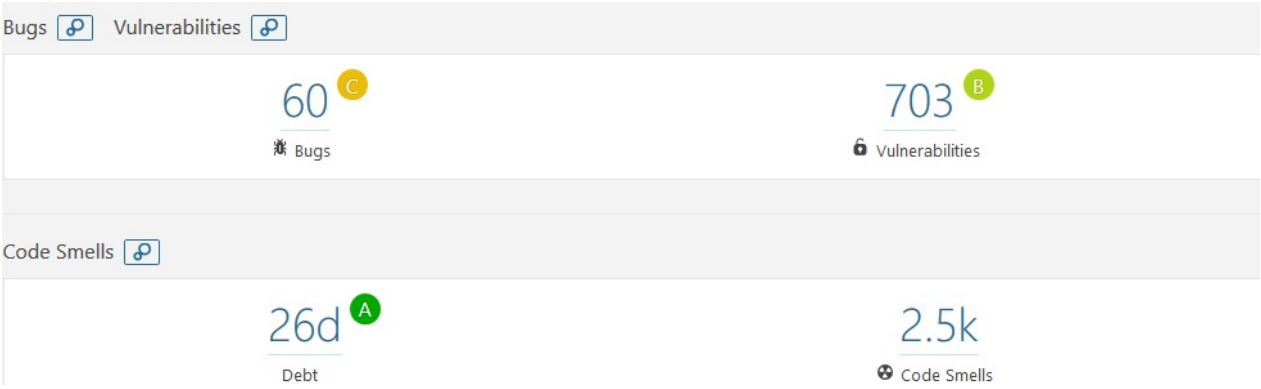


Figure 12 - *Welcome screen of SonarQube*

Issues found

Using ReSharper, 96 ToDos were identified. They are related to functionality, code quality or gameplay issues. Although it is good that developers are aware about these issues, such policy makes it more difficult to track progress in fixing these issues. These kind of issues should be managed by a centralised issue tracker (e.g. GitHub issues). Furthermore, it complicates comprehension of *osu!* structure and development strategy by new contributors. ToDos can be used as a short-term solution, when a developer is currently working on that specific section. However, the used ToDos should be removed before merging. This is needed to prevent ToDos from getting lost in the source code and being forgotten about. Moreover, several ToDos are rather unclearly described and open for discussion, which does not encourage resolving them.

Combining the four analysis of the tools, the most common issue is the amount of duplicated code. Code pieces with duplicated code occur mostly in legacy and test classes. To improve code quality it is recommended to refactor the latter classes. A solution for duplicate code is to extract the code to a separate method and call that method. In addition, it is unlikely that legacy code will be refactored if it does not contain known bugs.

Another issue found is the cyclomatic complexity inside several classes responsible for game logic with extensive switch statements. These extensive switch statements could be reduced with some refactoring by introducing a strategy or state design pattern.

Other remarkable issues are: access modifiers, virtual member called in construction, equality comparison of floating point numbers and assignments made in sub-expressions. These issues will most likely not crash the program, however they might be the source of unexpected behavior or decrease the ability to read the code.

Testing debt

Among the various tools used to analyze the technical debt, none of them could report any test coverage. This was either due to a failure or C# not being supported.

The main testing strategy of *osu!* relies on visual tests covering most of the visual aspects of the game. By going through parts of the game or interface, it can be verified that the behavior is executed as intended. In the next figure we illustrate an instance of visual tests.

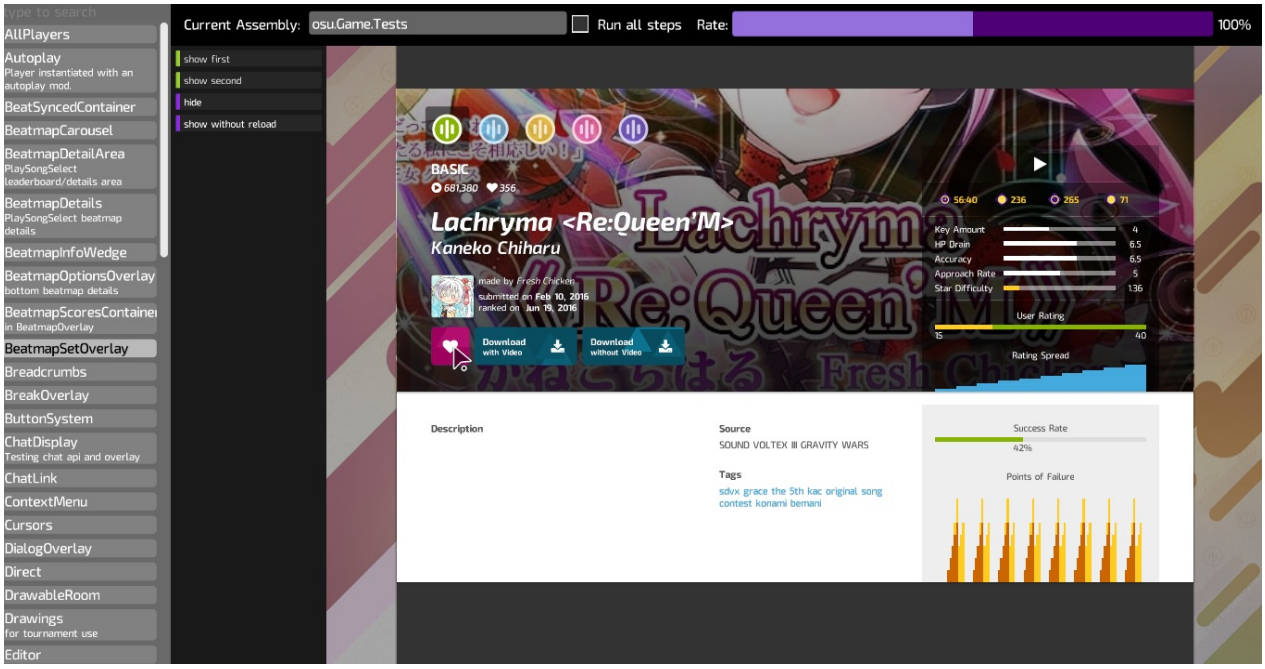


Figure 13 - _View of visual tests (leftmost column: list of components, second column: list of tests)

There are a few test classes in other packages that test the logic of the game, however their amount is limited and we could not manage to run them in Visual Studio. From AppVeyor, we can see that there are 146 passing tests. We can extract from the test names that they consist mostly of constructor tests.

For each project there exist a separate test package containing several test classes. Some of these tests are actual unit tests and some are helper classes for the visual tests, containing no asserts. Remarkably, in a recent pull request (#2075), they prepare the code for more testing.

With the current testing strategy, the game logic is only coarsely tested, and mainly by visual tests, which does not seem enough. Furthermore, the current testing strategy might explain the various issues detected by SonarQube for tests without any assert.

Documentation debt

Technical debt is not our only concern about this project. The documentation of *osu!* is very limited: most classes even lack any documentation. New contributors will find themselves struggling to answer simple questions: what should or can I do and where? Furthermore, no global development strategy is provided by the development team.

Social debt

The first thing that we noticed is that most of the design decisions, as well as the direction the development is taking, are made by *peppy*. Those who are not, are then often made by *smoogipoo*. While it is understandable *peppy* should lead the way "his" product evolves, this also leads to a very high bus factor: remove *peppy* (and perhaps *smoogipoo*) and the development would stop.

Furthermore, the design and development decisions are very scarcely communicated about: be it on GitHub or on the project's Discord channel, we could not find much regarding these aspects. It then only seems logical that most contributions would come from either *peppy* or *smoogipoo*.

Recommendation

We would recommend to start working on the documentation of the project. This includes code comments but also how other developers can help to improve and extend the application. This, to make it easier for new developers to contribute to the project and reduce the current bus factor.

Furthermore, they should look into their testing code and strategy, since part of the technical debt is inside these classes. Moreover, the project contains little amount of unit tests for the game logic, although the visual aspects of the game are well tested. These visual tests indirectly also covers several aspects of the game logic but minor mistakes can easily be overseen. Several improvements can then be made to reduce the technical debt.

Conclusion

With this chapter, we aimed to offer an all-around view on our open source project, *osu!*. First, we've seen that pure development is mainly driven by two people, while users and assessors of content are widely distributed around the globe for this worldwide online game. A tremendous amount of game-related content creation is left to the users: beatmaps, skins, and even seasonal backgrounds to add eye-candy at the game start.

Then, with the functional and development view, we detailed the key features of *osu!*, and how they are integrated in harmony as code. Three main projects allow a good separation of concerns: the framework is dealing with rendering and interaction, the resources contains the assets to use, and the main game implements the game logic and features.

osu! being a rhythm game where precision, readability and handiness directly affect the performance when playing, we took a closer look at usability. We found out that it has two main answers to that concern: first, it mimicks as much as possible the general layout of other rhythm games, to ensure a quick adaptation; then, it allows an impressive flexibility with many settings and the variability introduced by being able to apply a skin. Anyone can tailor the interface by downloading a skin, or even creating a new one.

As any software, *osu!* has technical debt. Although from our analysis it's not that important in terms of code smell, the lack of tests is more concerning. Moreover, the introduction for newcommers to participate in the code is rather lightweight, when not frankly sparse (in particular considering the lack of in-code documentation for some classes).

All of this make *osu!* an fascinating project to analyse, and we hope that this chapter has shown how rich this project can be.

We would like to thank:

- *peppy* for his feedback on our contributions,
- the osu! community as a whole to have answered some of our questions and participated to our survey,
- Gijs Weterings, Liam Clark, Romi Kharisnawan who did an excellent job as teaching assistant, providing us feedback and answering our questions,
- professors Arie van Deursen, Maurício Aniche, Andy Zaidman for setting up this insightful course.

References

1. Rozanski, N., & Woods, E. (2011). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.

Phaser - A Fun, Free and Fast 2D Game Framework for HTML5 browser games supporting Canvas and WebGL rendering.

Gerard van Alphen, Tom Catshoek, Tomas Heinsohn Huala, and Casper van Wezel.

Delft University of Technology

Abstract

Phaser is a 2D HTML5 game framework with support for WebGL and Canvas. It currently is the most starred Javascript game engine on Github and the repository is completely managed by the creator Richard Davey. In this chapter, the project is analyzed by looking at various aspects of Phaser. By creating a number of views and analyzing the stakeholders, (technical) debt and evolution, it was concluded that Phaser is a well-managed project with high standards for code quality. However, the project is lacking automated testing which leaves room for improvment.

Table of Contents

Introduction

Phaser is an open-source JavaScript game framework. It is licensed using the MIT License, so people are allowed to use it freely, even for commercial purposes. The framework implements both WebGL and HTML5 Canvas rendering, so it can be used in any browser with support for them. On top of that, by using 3rd party tools it is possible to package your Phaser apps as native ones.

The community around Phaser mainly consists of game developers who use the platform, either for hobby or commercial purposes. Phaser's main developer and integrator is Richard Davey. He has been working on Phaser since 2012, and it has grown tremendously since then. There are daily commits to the repository and every week a whole bunch of issues are created and closed.

Just a few months before writing this chapter, Phaser 3 was released. This new version improved a lot of all the problems they had with Phaser 2.

This chapter is written as part of the DESOSA (Delft Students on Software Architecture) book which summarizes all the work done for the TU Delft course on Software Architectures. It will provide an analysis of many different aspects of the Phaser project. This of course includes the technical perspective based on the code but also the business aspects are looked by defining the stakeholders.

Stakeholders Analysis

To get a feeling about everyone who is involved in the Phaser project, all the stakeholders will be listed and explained below. After that they will be combined into a Power-Interest grid.

Donators

The project has a Patreon page via which supporters can pledge a monthly contribution. This helps the funding of the project and its developers. The pledgers get some small rewards in return (a forum badge and a discount on new Phaser products). At the time of writing (22-02-2018) the Patreon has 171 pledgers, contributing a total of $1616 per month. There is also the possibility to do a one-off donation, for the people who do not like to commit to a monthly payment. The main sponsors of Phaser are CrossInstall and Orange Games two companies which use Phaser in their commercial products.

Communicators

Richard Davey, the creator of the project, is mainly responsible for the communication. However there is a large community which contributes training material, tutorials and knowledge on the Phaser forum and site. Richard is owner of the company Photon Storm Ltd which runs a HTML5 game development service.

Developers

At the time of writing (20-02-2018), there are 293 contributors to the project. The contributor top 5 (based on amount of commits) is:

Contributors	Notes
Richard Davey	Creator
Pavle Goloskoković	
Felipe Alfonso	Freelance programmer Photon Storm
Michael Hadley	
pnstickne	worked on first Phaser version in 2015

So the main incentive of the two main developers just arises from of their paid jobs. When looking at other people doing commits it is usually because they are a game developer using the platform themselves.

Maintainers

Richard Davey manages all pull requests and issues on GitHub. Therefore he is responsible for what reaches the production version.

Production engineers

Phaser is built using webpack, it uses several plugins to tailor the build process. Richard Davey and Rafael Barbosa both contributed to the webpack config.

Suppliers

Since Phaser is a framework for JavaScript browser games, the users have to supply a website where a browser can load the framework.

More important suppliers however are the dependencies of the sofware. Of course JavaScript is the most important one in that. All the rendering is handled by either WebGL or an HTML5 Canvas.

Node.js is the supplier of the `webpack` build system.

Support Staff

Within the Phaser community there is not really a separate group which can be marked as support staff. When a game developer has questions about the framework, the developers in the slack-channel or on the forum will usually try to clarify things.

Users

Of course the main incentive for Richard to start developing this framework is to use it for his own company Photon Storm. Besides, it is used by several other HTML5 game developers.

Competitors

With over 20 000 stars on GitHub Phaser is the biggest HTML5 game framework, but Phaser does have some notable competitors. For example PixiJS which is a lightweight library mostly used for the rendering part of game creation. In fact, Phaser used to be based on PixiJS, but in the meantime it has been heavily modified and incorporated in the system. Other interesting Phaser competitors can be found in this GitHub collection. Some of them focus on 3D games whereas Phaser's main focus is 2D games (although it has some support for 3D).

Power-Interest grid

We analyzed the power and interest of Phaser's stakeholders and visualized this in a power-interest grid.

The stakeholder with the most power and interest is obviously Richard with his company Photon Storm. He earns his money with Phaser and developing Phaser games and he manages the entire Phaser project.

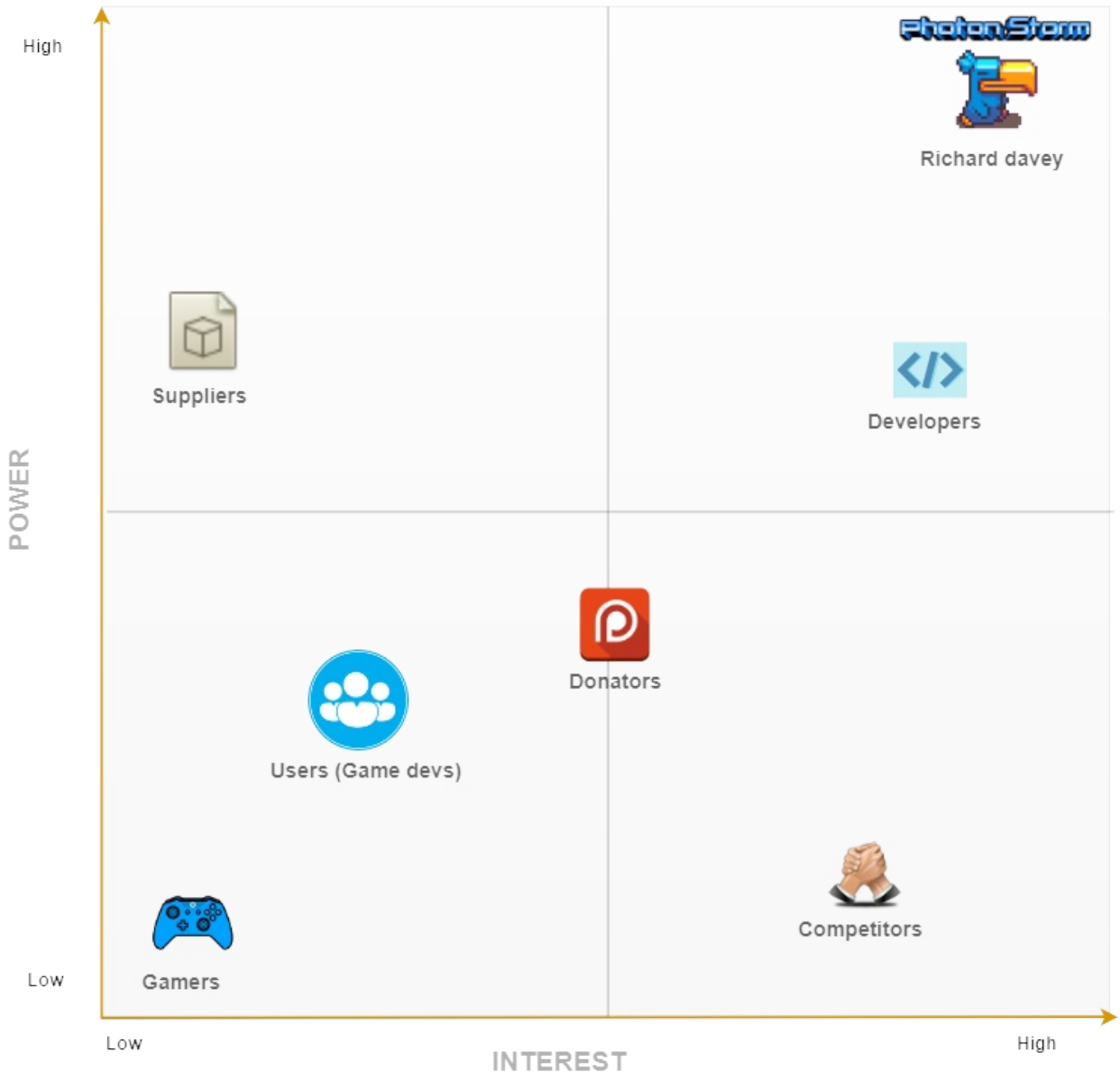
The Phaser developers also have significant influence in the project, but Richard will always have the final say.

Suppliers like WebGL have no immediate involvement with the project but Phaser does depend on them. Therefor they have high power but low interest. When for example WebGL decides to overhaul their API, this will require Phaser to rewrite the corresponding code.

The game developers use Phaser for their games so they have interest in the project, but not as much power. They can contact Richard for feature requests or bug reports. Donators have slightly more power and interest as they are willing to pay for the project on a monthly basis and (as stated on the Patreon page) they have a direct say on new features.

Competitors have relatively low power, as Phaser is by far the most popular framework. They do have high interest though, as they want to compete with Phaser.

The gamers, who play games created with Phaser have low power and low interest. For the gamer it does not matter as much which framework was used to create the game, the actual gameplay is what matters for them.



Context View

In this section the relationships of Phaser with its environment will be described, as per Rozanski and Woods. We will determine the system scope and responsibilities, analyze how it relates to the external entities involved, and what the interfaces between the system and those entities are.

System scope and responsibilities

Phaser is a JavaScript game framework which game developers can use to handle:

- Graphics rendering using WebGL and Canvas, mainly 2D but also with preliminary 3D support
- Animation, tweens and interpolation
- Sound effects and music
- Input from keyboard, mouse, touch, and gamepads
- Asset loading from URLs
- Physics using Arcade physics, Matter.js, and in the future P2 Physics and Box2D

For the sake of completeness, there are also things Phaser does not do:

- Implement game logic

- At the time of writing, implement full 3D graphics support
- Package itself for environments other than browsers
- Host game assets

External entities

Since Phaser is a framework, it certainly does not operate in a vacuum. Several of its relations to the outside world are described below:

- It is developed using JavaScript and HTML5
- It runs in most modern browsers, like Firefox and Chrome
- Games made with Phaser can be packaged for Android, iOS and as a native app using 3rd party tools like cordova and electron.
- It uses a custom rendering engine which supports both WebGL and HTML5 Canvas
- Webpack is used as a build system
- GitHub is used as a version control system and issue tracker
- Development of Phaser is financed by its Patreon, Paypal donations, and its two main sponsors, OrangeGames and CrossInstall.
- It is available under the MIT license
- Game developers build their games on Phaser
- Gamers of all ages play the games made with Phaser

External interfaces

Here we will describe the interfaces between Phaser and its external entities.

Entity	Data	Service	Event
JavaScript	both	both	provider
HTML5 / Canvas	consumer	provider	x
WebGL	consumer	provider	x
Webpack	x	provider	x
GitHub	x	provider	x
Games	x	consumer	x

Some of this might require a little clarification. We see JavaScript as both a data provider and consumer as it can be used for asset loading and all other forms of data input, as well as output (e.g. uploading high scores or save files to an external server. That would be up to the games made with Phaser to implement though, but the option is there). We also see it as both a service provider and consumer as it is obviously used to call the Phaser api, but Phaser itself is written in JavaScript. On top of that, we also see it as an event provider, as it passes input events from input devices to Phaser to handle later.

HTML5, or more precisely the Canvas element, is seen as a data consumer, as Phaser passes information to it on what to draw. It is also a service provider, as it provides drawing functionality to Phaser. The same goes for WebGL.

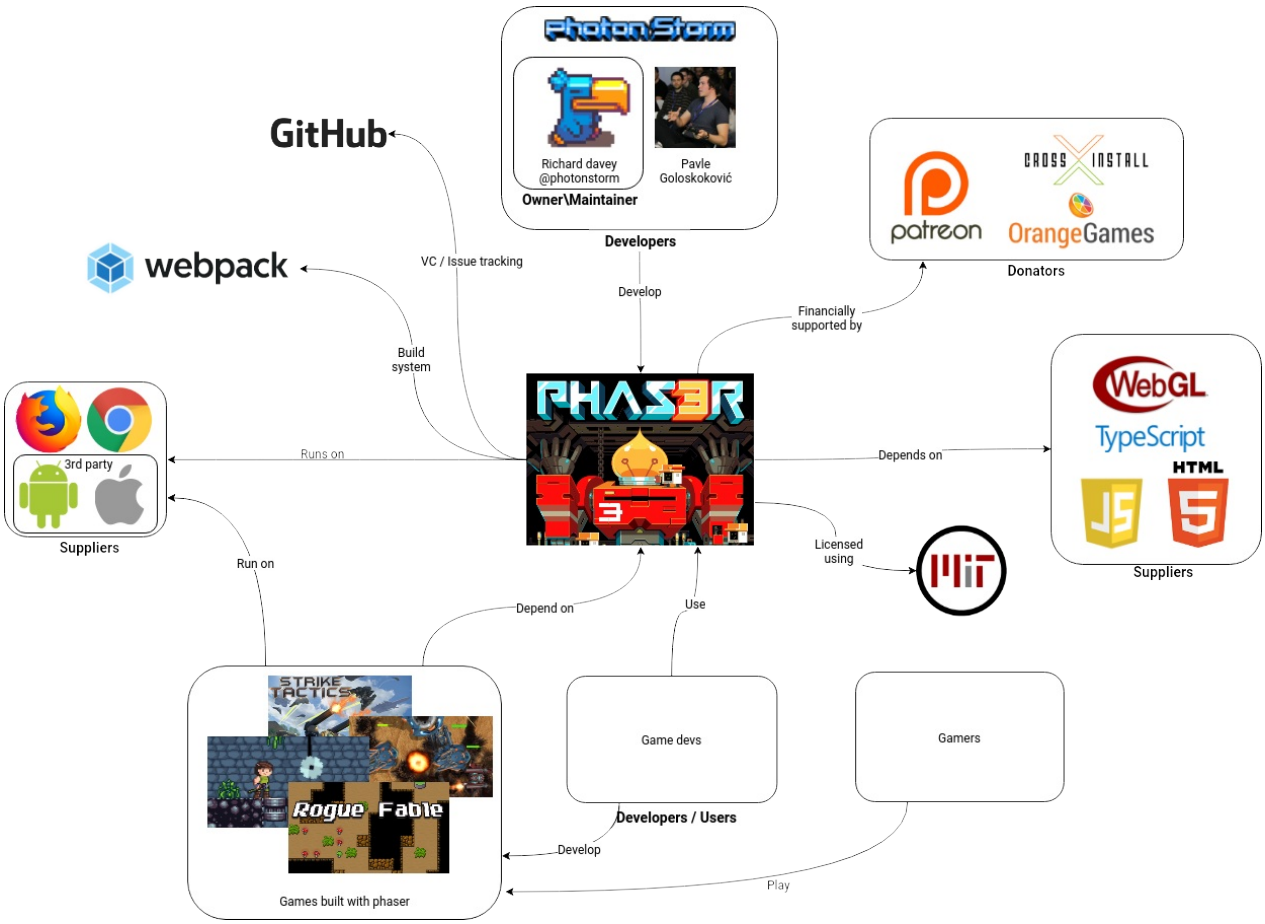
Webpack provides a module bundling service to Phaser, which is used to create a distributable, and possibly minified build of Phaser to use in production.

GitHub provides version control and tools for collaboration.

Games consume the services provided by Phaser by calling its API to do all things games want to do.

Context diagram

An overview of Phaser and its relations to the external entities mentioned above can be seen in the Figure below.

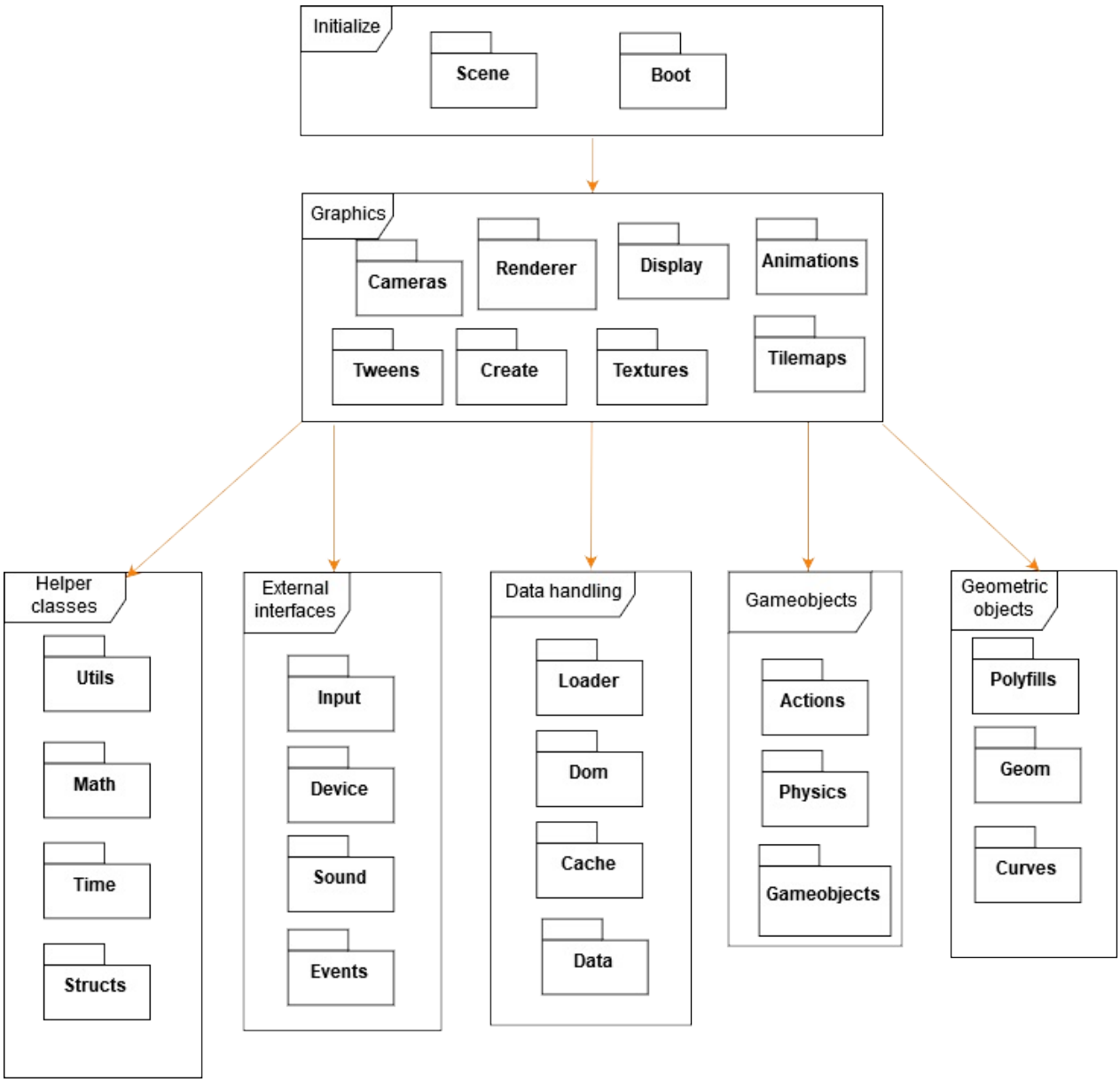


Development View

Since software development environments often require special dependencies or configurations, this section covers important details of the Phaser development environment. The organization and structure of the code and architecture will be discussed, as well as the testing facilities.

Module Organization

The Phaser project consists of 28 different packages and some configuration files. The most important package is the `boot` package which initializes the game. When inspecting the project, we saw that all these packages can be grouped as modules. An overview of the modules present in this project is shown in the Module Structure Model below.



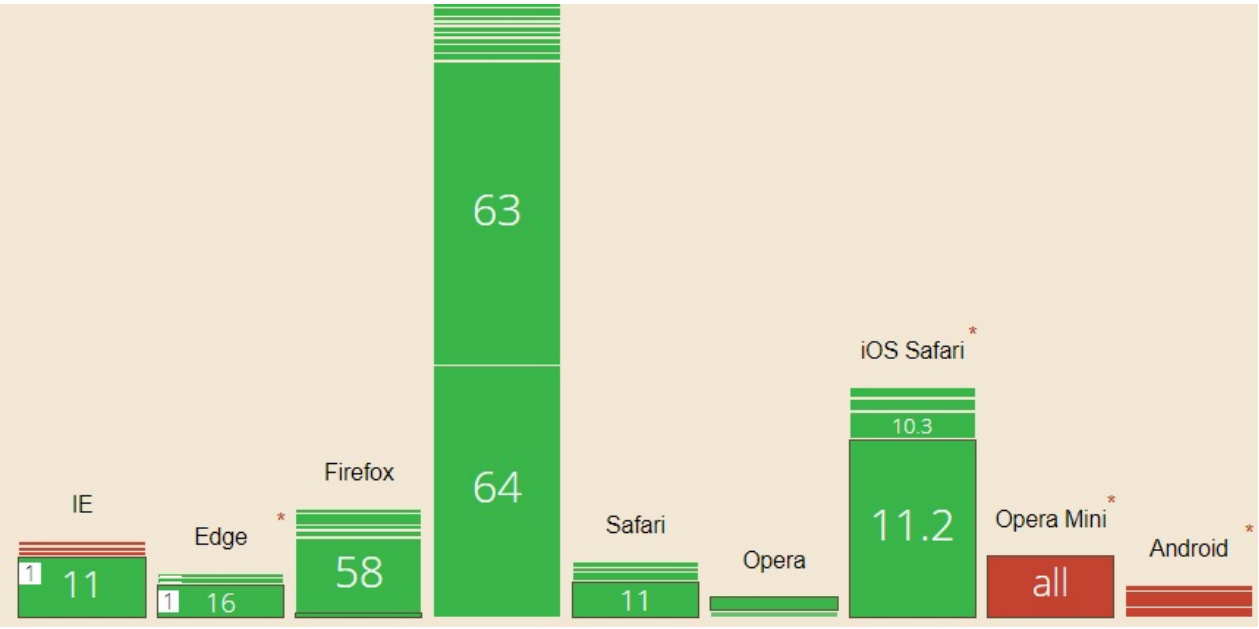
At the first layer of the Phaser project we saw that there was an initialization module. This module contains the `boot` and `scene` packages. The main package `boot` is responsible for setting up the game including external resources. The `boot` package also sets up the scene manager which is located in the `scene` package. This package is responsible for handling everything that is related to the visual aspects of the application.

Secondly, we have the graphics layer which is launched by the `scene` package. This layer is responsible for visualizing the graphical content in Phaser. The main package here is the `renderer` package, the renderer is responsible for managing all visible objects on screen. More on the renderer is explained in next section.

The third and final layer contains five modules, they are mostly helper functions for the rest of the framework. The `GameObject` module and `geometric objects` module handle all the different kind of objects to be represented in the game. For instance, images and text objects are GameObjects here, these objects can be adjusted with the help of the `physics` and `actions` package. The data module is responsible for handling all the incoming data used with Phaser. Furthermore, the external device module is responsible for managing and connecting all external inputs (software and hardware wise). Finally, there are the helper packages which contain the basic functions needed in this framework. The most important one here is the `math` package as maths are used a lot in the interaction (rotation, translation etc.) with graphical objects.

Common Processing

Because our framework supports multiple platforms there are some interfaces in the software which open up the possibilities for using different implementation. The biggest example for that is the Renderer. The framework has a `Canvas` and `WebGL` Renderer which can be used interchangeably so when a device/browser does not support `WebGL` the framework automatically falls back to the `Canvas` Renderer. So this is not really a Common Design Model, but more a Common Behaviour Model just doing Common Processing. This is useful when you want your game to be compatible with for example older versions of Internet Explorer or native Android browsers, as some of them do not support WebGL:



Source, red boxes indicate browser versions that do not support WebGL

You can specify which renderer you want to use with the `type` attribute in your game config. The options are:

- `type: Phaser.AUTO`, this will automatically detect which renderer is supported and choose the best option
- `type: Phaser.WEBGL`, for WebGL rendering
- `type: Phaser.CANVAS`, for Canvas rendering
- `type: Phaser.HEADLESS`, for no rendering at all

The big advantage of this design is that the implementation is the same for each renderer, which means you only have to write the code once to support multiple renderers/browsers.

Standardization of Design

The `GameObjects` module contains a lot of different classes (e.g. `container`, `group`, `images`, `mesh`, `particle emitter`, `sprites`, `text`, `tilemap`). Each of these classes behave similarly, so for example the action `translate(x,y)` can be called on any of those objects in order to translate them. This is standardized by having all individual GameObjects extend the `GameObject` class.

Using this standardization, all of these GameObjects can then be created in a Scene and be manipulated in the same way. Any additional functionality can be implemented in each individual `GameObject`. This makes the system more maintainable and easier to extend. Whenever a new `GameObject` needs to be added, simply create a class which extends `GameObject` and implement the additional functionality.

Instrumentation

The framework has an `DebugHeader` to provide useful feedback to the developer by means of console logs and statistics.

Codeline Organization

The codeline organization of a system is all about the structure of the code base itself and how the project is managed in terms of releases. The code base structure of the project is not very special, besides the regular git folders and required configuration files. There is one `src` folder containing the packages as already showed in the model structure model.

As mentioned, since recently the project has a working CI environment. Every commit is built by Travis to reduce the risk of releasing failing code. Dependencies are managed using the package manager from Node.js (`npm`). The code itself is then built using `webpack` , which bundles the source code in a single JavaScript file which can be used in the browser.

All releases of Phaser are managed on GitHub. Whenever a new release is ready, Richard will tag a commit with the version and update the changelogs. These changelogs contain detailed information on what has been changed and by who this has been changed. The time between these releases is about 5 days on average

Technical Debt

In this section we investigated the technical debt of the Phaser repository. Technical debt is all about how much it would cost extra in the future if you have to redevelop a solution which was chosen now, instead of applying a better solution now that would take more time. So technical debt concerns the code quality of a software project and if this code was tested properly. To analyze the technical debt, we use code quality tools like SonarQube to get an overview of the source code quality to detect pieces of software that could be improved.

Code quality tools analysis results

We chose to use two different online code quality tools: SonarQube (we used SonarCloud as online platform for it) and DeepScan. They both gave roughly the same output, the results of both scans are discussed in the following sections.

Scan results

SonarQube works with 4 categories to measure the code quality. The first category scans the project for bugs and vulnerabilities, where bugs are related to the reliability and the vulnerabilities are related to the security of the system. The initial scan resulted in the following amount of bugs/vulnerabilities:

The bug scans looks for parts of code that could fail, for example possible null references or syntax errors. It identified some typo's, some cases where a variable could be null or undefined when it is used. About 19 of the 35 bugs are related to the potential unintended use of bitwise operator `&` instead of conditional `&&` . This bitwise operator appears in a if-condition usually in combination with a mask of some sort resulting in a number. This number is then implicitly compared by JavaScript to see if it is zero (false) or something else (true). For readability it might be good to change this to an explicit comparison like `(a&b) != 0` , but this depends on the type of developer and their experience with masks.

Furthermore, the three vulnerabilities it found were all of the same kind, namely: `Review this "Function" call and make sure its arguments are properly validated.` . We were unfamiliar with this vulnerability, but fortunately SonarQube provides an explanation for each scan result which in this case was:

> In addition to being obtuse from a syntax perspective, function constructors are also dangerous: their execution evaluates the constructor's string arguments similar to the way `eval` works, which could expose your program to random, unintended code which can be both slow and a security risk.
>
> In general it is better to avoid it altogether, particularly when used to parse JSON data. You should use ECMAScript 5's built-in JSON functions or a dedicated library.

So as these function constructors work similarly to `eval` , a string input can be evaluated to JavaScript code. One of the those Function constructors in Phaser's code is:

```
new Function('a', 'return {minX: a' + format[0] + ', minY: a' + format[1] + ', maxX: a' + format[2] + ', maxY: a' + format[3] + '};');
```

So whenever a user is able to manipulate the contents of the format array (for example through input in the game), he will be able to execute arbitary code. It is unlikely that this will be a security risk, but it might crash the game or be exploited to cheat the game. Therefor it is better to avoid them altogether.

SonarQube estimates that the time required to fix the bugs is around 4 hours and the time to fix the vulnerabilities is about 15 minutes. SonarQube also analyzes the technical debt and code smells:

This is a very positive result, a technical debt of only two days. The code smells include redundant and unused code, confusing code and more. By quickly analyzing these code smells it becomes clear that most of these are related to useless assignments or unused variables, so fixing these code smells will not cost much effort. The most of the other code smells were related to boolean expressions which always seem to evaluate to true.

Fixing these code smells will improve the maintainability of the system, as it will make the code more readable and clear it from redundancies. In general the code smells are not bugs, the code still functions with the smells in it. However, they increase the risk of introducing bugs in the future which is why it is recommended to keep the code free of smells. For example, a piece of unused code could be triggered after a refactor which might cause all sorts of issues.

Another aspect SonarQube looks at is test coverage:

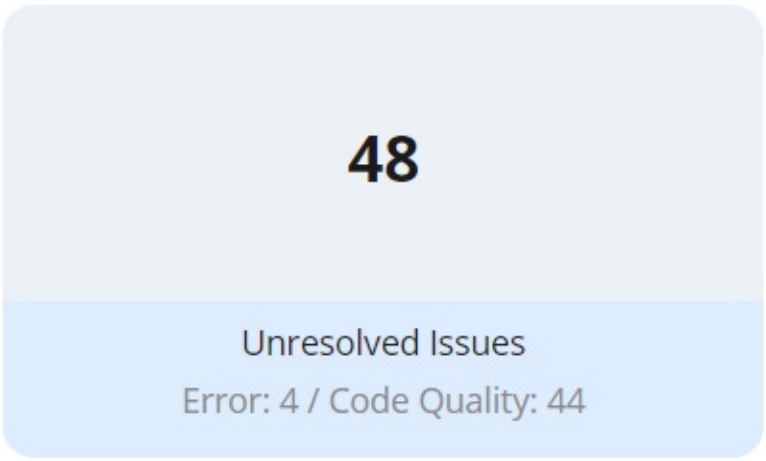
This is where the project really lacks behind; it simply does not have any tests. More on this can be found in the Testing debt section.

The final aspect SonarQube check is code duplication:

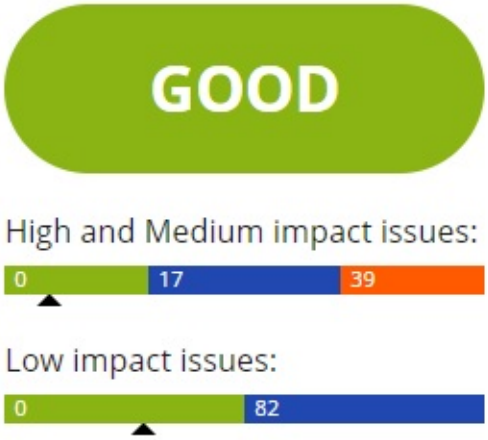
Obviously code duplication is generally bad practice as it makes the system more difficult to maintain. Fortunately Phaser has very low code duplication. In fact, the two files with the most duplication are debug classes, which are not part of the actual system.

The other tool we used was DeepScan, a code analysis tool specifically for JavaScript.

DeepScan was able to find a total of 48 issues:



Based on the amount of issues, DeepScan provides a code quality grade, which turned out to be "Good" :



The grading is explained in DeepScan's official documentation.

Testing debt

The project does not contain any automated tests, apart from Travis CI which was enabled again recently. This just checks if `webpack` and `ESLint` do not return any errors. And it has to be noted that `webpack` just checks for JavaScript syntax errors, so it is still possible for errors like typos to pass the CI. The extensive example repository can be used to manually check each part of the system using small pieces of example code.

It could be argued that it is hard to test a gaming framework, which is partly true. For example, how would you test that for example a certain shape is drawn on the screen as expected? This however, is not an excuse to not have any tests at all. Unit tests could be used to test the logic of the code. The project would benefit from this, as it could assure that for example the wide variety of calculations done by the framework are correct.

A good example of a bug, which could have been spotted earlier with these tests, is a matrix rotation function which rotated the matrix in the opposite direction of what it was supposed to, see this commit. Tomas discovered this and informed Richard about it. Richard then fixed it and thanked Tomas for the find.

When investigating ways to test a JavaScript game engine we stumbled upon the Crafty game library. This project is tested with a JavaScript unit testing framework called QUnit, which could be a useful addition to the Phaser project.

As Phaser makes use of a lot of helper functions, there is a need to start testing these functions first. For instance, the Math package contains helper classes for performing mathematical operations and the Utils package contains additional functionalities for the objects String, Array and Object. Furthermore, it would also be beneficial to test the data loading, as the graphics used in Phaser projects could consist of external files like images or gifs. Another part which could use some testing is the GameObjects module as this contains a lot of configurations options, as well as a wide variety of actions that can be performed on these objects.

The extensive collection of examples for the project serves both documentation and testing purposes of course. However, setting up a real testing framework to also test the rendering could improve the quality and reliability of Phaser as a framework.

The lack of testing was also mentioned in a recently created issue by one of the contributors(#3361). Here it is being stated that the phaser3-examples should and could be used for regression testing, but that the actual implementation of this is still to be reviewed and discussed. We contacted Richard on Slack to ask him about his thoughts on testing, to which he answered:

> I guess tests could be made for the non-visual parts of the API, although to be honest those are usually the ones that break the least.

A nice way to test the visual part is to take screenshots of the examples and compare them with a reference screenshot. All the non-deterministic things like random numbers will pose a problem here though. So to really hit this off, a deterministic version should be created. And since most of the Phaser 2 examples are not working with Phaser 3 now, it would have been wise to start with this during Phaser 2 already so it would be easy to see the status of the switch to Phaser 3. This wish of having these features implemented in the past is of a textbook example of technical debt!

Debt evolution

As we have seen with the help of SonarQube, there were a lot of code smells related to useless assignments or unused variables. When looking at this kind of code smell it becomes clear that almost all of them were introduced in the last six months. This is probably related to the fact that the last two years were all about going from Phaser 2 to Phaser 3. When we analyzed the technical debt of an earlier version of Phaser 2, we saw that this version had a technical debt of 23 days. Most of these code smells were related to empty statements and the reuse of variable names in the same class. Anyhow, these code smells were not similar to those of the current technical debt. This is due to the fact that Phaser had been refactored before Phaser 3 was released and by doing this they introduced new code smells. As a lot of these code smells were about unused assignments, it is possible that these variables were actually used in the Phaser 2 code.

Technical debt discussion

At the time of writing Phaser's code contains 24 TODO's and no FIXME's. The TODO's can be considered as a means to communicate technical debt. Debt is also discussed by developers on the Phaser forum and in GitHub issues. Developers create an issue when they find a bug, or ask questions about the system. For example as mentioned earlier, a user was wondering why the project is lacking automated test and opened an issue about it. So some of the people do find it important, but before any progress is made on this topic the owner of Phaser still has to decide what to do with it.

Functional View

In this chapter we will look at Phaser from a functional point of view. We will define what the system is required to do and what modules are used to achieve this. We will also look at the external interfaces Phaser exposes and how these can be used.

Capabilities

What follows is a non-exhaustive list of Phasers top level modules and what functionality they provide. These were considered the most relevant to users of Phaser.

Module	Functionality
Phaser.Actions	Apply actions to game objects
Phaser.Animation	Provide animation functionality to game objects that support it
Phaser.Cameras	2D and pseudo 3D camera functionality
Phaser.Curves	Math functions related to curves and paths (Cubic/Quadratic Bezier, Spline etc.)
Phaser.Game	Phasers main module responsible for setting up all subsystems and running the game loop
Phaser.GameObjects	Provides builders for all game objects
Phaser.Geom	Functions related to geometric primitives
Phaser.Input	Keyboard, mouse, gamepad and touch input
Phaser.Loader	Asset Loading
Phaser.Math	Math utilities and functions
Phaser.Physics	Arcade, Impact and Matter.js physics
Phaser.Renderer	Canvas and WebGL renderers
Phaser.Scene	Game "world" containers, can be seen as states
Phaser.Scenes	Scene manager, plugins, settings and systems
Phaser.Sound	Sound management and playback, HTML5 and webaudio support
Phaser.Textures	Texture frames, sources and management
Phaser.Tilemaps	Creation of static and dynamic tilemaps
Phaser.Time	Time related functions and events
Phaser.Tweens	Tween building and management
Phaser.Utils	Array, Object and String utility functions

External interfaces

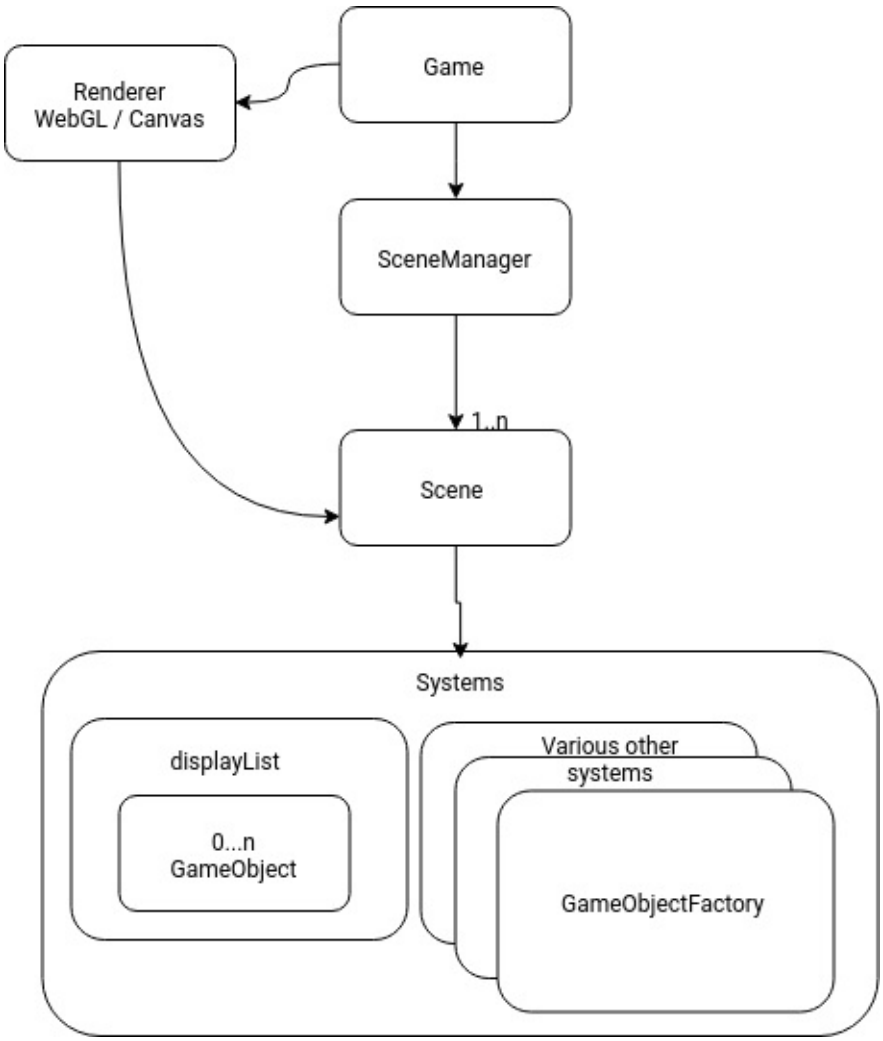
All modules mentioned above are exported as external interfaces and can be accessed in any project which uses Phaser. However, in a typical game they might not all be interacted with directly. Since Phaser gives you a lot of freedom in the way you implement your game, we will go over a greatly simplified example representing a typical use case.

- To initialize the game, Phaser.Game is called with a config file to set up the managers and the game loop
- In this config a scene is defined, which has preload, create, and update functions
- Preload runs, loading in assets from URLs
- Create runs, which sets up contents of the scene and creates game objects
- Then, every game loop update is called as long as the scene is active
- Update handles input, updates game objects, handles game logic, and more.

So, the way most users will interact with Phaser is through defining scenes, in which most functionality is handled by Phasers underlying systems. Of course users still need to implement their game logic themselves. It is possible to have multiple scenes, even running in parallel. For example, this allows you to keep logic for an inventory management screen and the actual game separated.

Internal structure

The internal elements which make up Phaser largely correspond to the modules above. To make more clear how they relate to one another, please see the diagram below.



Every iteration of the game loop, the following happens under the hood:

- the scene manager is told to update all the scenes it contains
- the scene manager is passed the games renderer and told to render all scenes

Conceptually this is pretty simple, but code wise it is pretty hard to follow. Especially since the documentation of Phaser 3 is still pretty incomplete.

Evolution of Phaser

In this section the Evolution of Phaser of the past years will be discussed.

Since the start of the Phaser framework in 2013 a lot has changed of course. The performance of both mobile and desktop platforms have improved even further and that a lot of new possibilities were created by different frameworks. These frameworks allowed the growth of Phaser. WebGL is a good example of this, because WebGL provides a standard rendering API which works across most browsers these days. PixiJS is a graphics library build on top of WebGL which allows for easy rendering of certain objects. Another nice feature about PixiJS, is that it automatically falls back to HTML5 Canvas rendering if WebGL is not available.

However because Phaser was extended more and more, there was a need for more freedom and options during Phaser 2. This is the reason why PixiJS was incorporated in the Phaser project itself when the switch to Phaser 3 was made. So the whole idea of PixiJS is now fully embedded within Phaser itself. Since both projects make use of the MIT License, it also allowed the Phaser 3 developers to directly copy the PixiJS code into the Phaser codebase.

Since the changes made since Phaser 2 are so drastic, most of the documentation that was created for the Phaser 2 API is not valid anymore. Just as with many other projects, documentation here is always lagging the deployed codebase. There is a separate Phaser 3 Documentation repository, but it is not yet hosted on the Phaser website since it is not yet complete according to the Richard. Since all

documentation is automatically anyway, it would have been nice to just publish it anyway and refresh it every release because this might trigger other developers or users into updating the documentation when they need to.

Conclusions

In this chapter we analyzed the Phaser HTML5 game framework. We can conclude that this is a well-functioning and thought-out project.

The first section describes the stakeholders involved in this project. The creator of the project, Richard Davey, is by far the most important stakeholder. He manages the entire project on his own. We did identify several other stakeholders, like donators and the game developers/users. Furthermore we analyzed the power and interest of those stakeholders, where we concluded that Richard (with his company Photon Storm) has the most power and most interest. We also looked at issues and pull requests to determine the influence of stakeholders and analyze the integrators.

In the context view section we analyzed the dependencies of the project and visualized this in a context diagram. Here it became clear that depends on a wide variety of external entities, like browsers and suppliers.

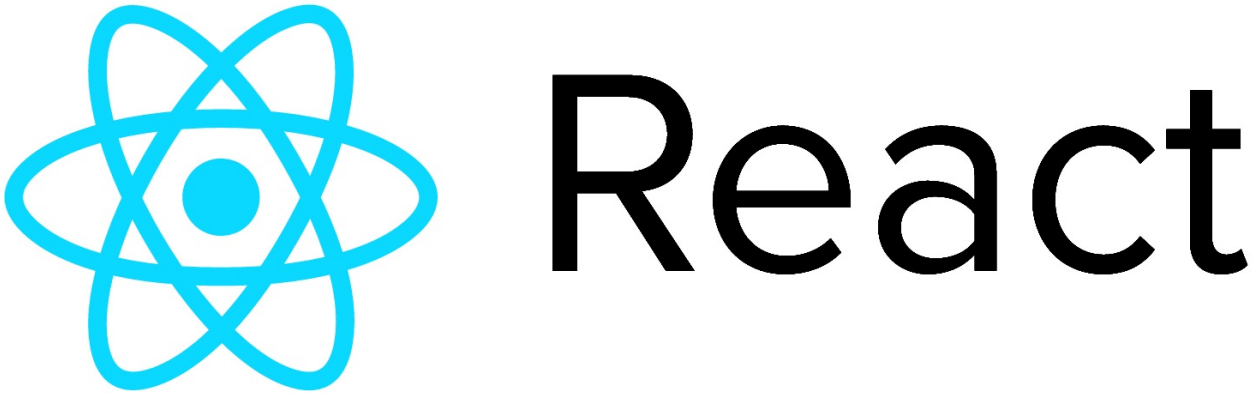
Phaser uses common processing (for example the renderer) and standardization (for example the `GameObjects`) in the design of the project. In the Development view section we looked into the structure of Phaser's codebase. We also concluded that there was no automated testing, apart from the linting in Travis CI and the provided examples which can be run manually.

The technical debt turned out to be very low, only two days. We ran automated code quality analysis tools (SonarQube and DeepScan) to conclude this. Most issues were minor so they are relatively easy to fix. However, as there are no test, the testing debt was very high.

Phaser keeps evolving with releases almost every week, with a a major release only recently, Phaser 3. We managed to make some contributions which helped progress the project. All in all, Phaser is a really interesting project which is professionally managed, albeit by a single person. We enjoyed working on the project and will follow the progress it will make over time.

References

1. http://phaser.io/
2. Rozanski, N. Woods, E. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012.




David Alderliesten


Jesse Tilro


Floris Doolaard


Olivier Maas

Abstract

React is a JavaScript library that provides a declarative, component-based framework for developing interactive user interfaces that update in real-time without requiring page refreshes. We analyze the *React* project from several angles. First, we provide an overview of the stakeholders involved and their relative power and interest. Next, in the context view, we look at all the external tools and platform that *React* makes use of for development, testing, deployment, and so forth. We then move on to the development view, where we describe *React*'s internal structure as well as some of its standardized development practices. Finally, we analyze in detail the technical status of *React*: We identify its technical debt, including how it evolved and which associated discussions took place.

Table of Contents

1. Introduction

The *React* framework, sometimes referred to as *ReactJS* or *ReactFB*, provides a declarative, component-based framework for the development of interactive and real-time user interfaces. Whereas web applications commonly require refreshing to view updated information, *React* allows for the manipulation of a virtual domain object model that interacts with the *real* domain object model of the browser, allowing for components which directly update when data is changed. As a result, *React* provides a web application user interface which changes in real time, causing greater interaction between the user and the application.

To assist developers and in an attempt to contribute to the project, this chapter focuses on the architectural aspects of *React*. These include a stakeholder analysis, a development viewpoint of the project, contextual analysis, and other information. Together this information provides an insight into the *React* project, possibly leading to a clearer overview of the project for new developers or to identify areas in which improvement is possible to ensure a greater degree of maintainability in the future.

2. Stakeholders

The stakeholders section aims to provide an overview of the various types of stakeholders of the *React* project, alongside analysis of their power and interest positions and their roles, as defined in Rozanski and Woods' '*Software System Architecture: Second Edition* [1].' Furthermore, a power-interest diagram is provided to indicate relative stakeholder influence regarding their control over the *React* project and their level of interest.

2.1 Stakeholders by Category

2.1.1 Acquirer

Within the context of *React*, the acquirers consist of Facebook, Inc., and all its subsidiary companies, including Instagram and Atlas [2]. Although development is open to (and relies on) open source developers, all contributors must agree to the Facebook contributor license agreement (CLA) [3]. Thus, Facebook continues to authorize the development and control of the project.

2.1.2 Assessor

The main assessor of *React* is Facebook. Although the project is maintained and assessed by Facebook, it heavily relies on the open source community for continued development. To facilitate this, *React* has been made open source and allows for implementation of the project as a dependency solely through the utilization of the *MIT* license agreement [4], which allows both private and commercial utilization of the *React* software, including modification and distribution, but waives liability and warranty requirements away from Facebook, thus ensuring the company cannot be legally held accountable for damages due to *React* software.

To further ensure compliance with legal requirements, all dependencies upon which *React* depends must also feature similar license terms to ensure waived liabilities and allow for distribution of the dependency within another project or package. Major dependencies within the *React* repository include 'Babel', a JavaScript compiler; 'ESlint', a *linting* utility for *JavaScript*; and *Rollup*, a module bundler for *JavaScript*. A complete list of dependencies can be found in the project itself [5].

2.1.3 Communicator

Based on the repository activity, the communicators dealing with pull request discussions and oversight are:

- Dan Abramov (@gaearon [7])
- Brandon Dail (@aweary [8])
- Brian Vaughn (@bvaughn [9])

These three communicators account for a majority of the active discussion and reviewing of pull requests and the technical communication of the *React* architecture.

2.1.4 Competitor

Stakeholders which are competitors include:

- *VueJS* [14]
- *Angular* [15]
- *Elm* [16]
- *CycleJS* [17]
- *Preact* [43]

2.1.5 Developer

When analyzing the repository in the current state, there are a total of 1,157 minor developers, and the following ten major developers based on their contributions, commits, and features added:

- **Sophie Alpert** (@sophiebits [19])
- **Paul O'Shannessy** (@zpao [20])
- **Dan Abramov** (@gaearon [7])
- **Sebastian Markbage** (@sebmarkbage [21])
- **Andrew Clark** (@acdlite [22])
- Pete Hunt (@petehunt [23])
- **Brian Vaughn** (@bvaughn [9])
- Cheng Lou (@chenglou [24])
- Christopher Chedeau [25])
- **"Jim,"** (@jimfb [26])

These are the main developer stakeholders. Developers that are bolded are moreover organizational developers. Developers also hold roles related to providing support to end users, maintaining testing and engineering requirements, and acting as testers and verifiers for implemented behavior and changes. The developer is the stakeholder that, although not necessarily having the most power within the organization structure, performs the most versatile set of tasks. All developers must sign the *Facebook Contributor License Agreement* (CLA) and agree to the terms set out in the contribution guide for *React* [18].

2.1.6 Maintainer

A maintainer is a stakeholder that ensures documentation and knowledge is preserved over time. The responsible stakeholders for this task is the core *React* team and all its associated engineers, since they maintain the official *React* site and documentation. This core team published a subset of their meeting notes within a separate repository (*React-Notes*) for documentation purposes [42].

2.1.7 Supplier

The React team makes use of *Github* to host the project and coordinate with other developers, and the project package is hosted on and distributed via npm's package registry, npm-registry.

2.1.8 Tester

A tester is a stakeholder that ensures that *React* works correctly by performing a combination of unit tests, smoke tests, and other procedures both automated and manual which verify correct behavior. The role of testing is performed in two segments. Manual testing is done to verify behavior, which is the responsibility of all developer stakeholders and contributors who developed within the *React* repository.

The second type of testing stakeholder are the automated testing tools. The official *React* repository relies on *CircleCI* [27] and *Coveralls* [28], who thus become two testing stakeholders. These stakeholders ensure that the automated tests, of which the repository boasts 90% coverage, pass and the desired functionality is maintained throughout development.

2.1.9 User

Some high-profile users, which consist of websites that utilize the *React* Library and are known to have significant traffic or awareness in both technical and non-technical communities, include:

- Netflix [29]
- Yahoo! [30]

- Airbnb [31]
- Discord [32]
- React Native [33]

Regarding these users, it should be mentioned that *React Native* is an implementation that allows native mobile applications to be developed. This means, in a simplified manner, that *React* is utilized to build components which are then compiled natively as opposed to being active on a server. *Facebook*'s subsidiaries, such as *Instagram* and *Oculus VR*, are also excluded from this category, as they fall under the greater umbrella of *React*'s main developer --- namely, *Facebook*. These stakeholders have a greater stake within the project than mere utilization of *React*, and due to this they are excluded from the user stakeholder role within this analysis.

2.2 Stakeholder Influence

To measure stakeholder influence and to dictate the level of communication a stakeholder must receive regarding *React*, a power-interest matrix was constructed. This matrix indicates stakeholders of increasing interest on the *x*-axis, and stakeholders of increasing power upon the *y*-axis. A stakeholder that is high power and low interest must be kept satisfied and informed only when necessary, whereas a stakeholder with low power but high interest should be informed and kept up to date, but not necessarily catered to. Stakeholders with both significant interest and power must be maintained closely, as they steer the project. The overview in figure 2.1 contains the most active contributors and most influential organizations.

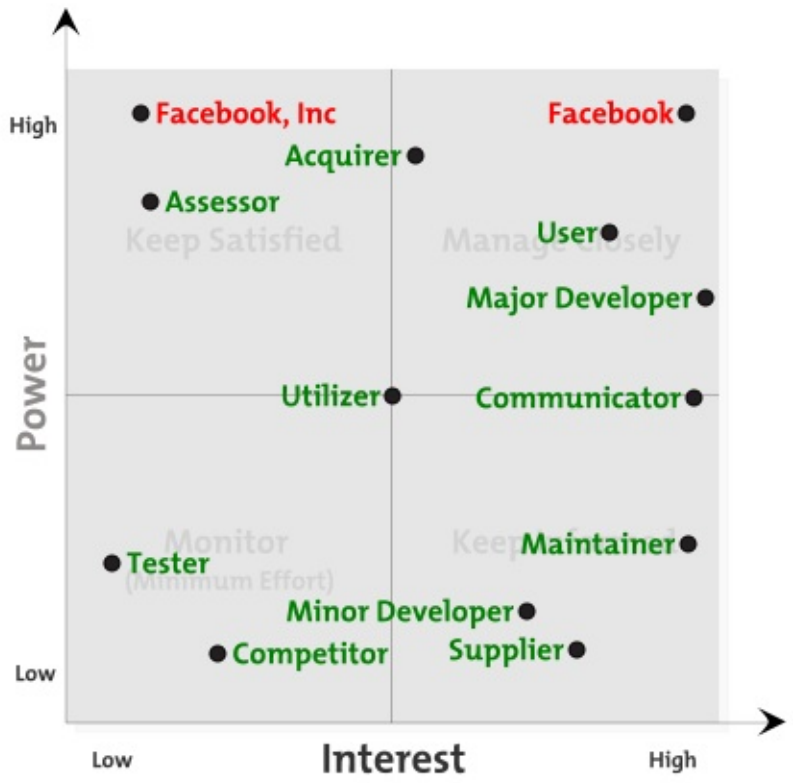


Figure 2.1: a diagram showing stakeholder power versus stakeholder interest.

3. Context View

The context viewpoint describes the relationships, dependencies, and interactions between the system and its environment. This section identifies and discusses concerns regarding the system scope and responsibilities, external entities and services, and data used. As an illustration of the coherence of the entities a context diagram is also provided. The context view also includes other systems, organizations, or people that are involved with, or interact with, *React* in some manner.

3.1 System Scope

The system scope defines the main responsibilities that *React* provides, which consists of providing a *JavaScript* library for the development and utilization of user interfaces (UIs). This responsibility can be divided into the following sub-responsibilities:

- Allows users to render input data and output the data to the screen.
- Allows users to build encapsulated components that can manage their own state.
- Provides users with the flexibility to interface with other libraries and frameworks.

3.2 Context Model

The context model, as shown in figure 3.1, depicts the many associated external dependencies related to *React*. They have been grouped where possible and the arrows represent what kind of relationship the entity has with *React*. A quick summation is made whereafter the external entities will be described in more detail.

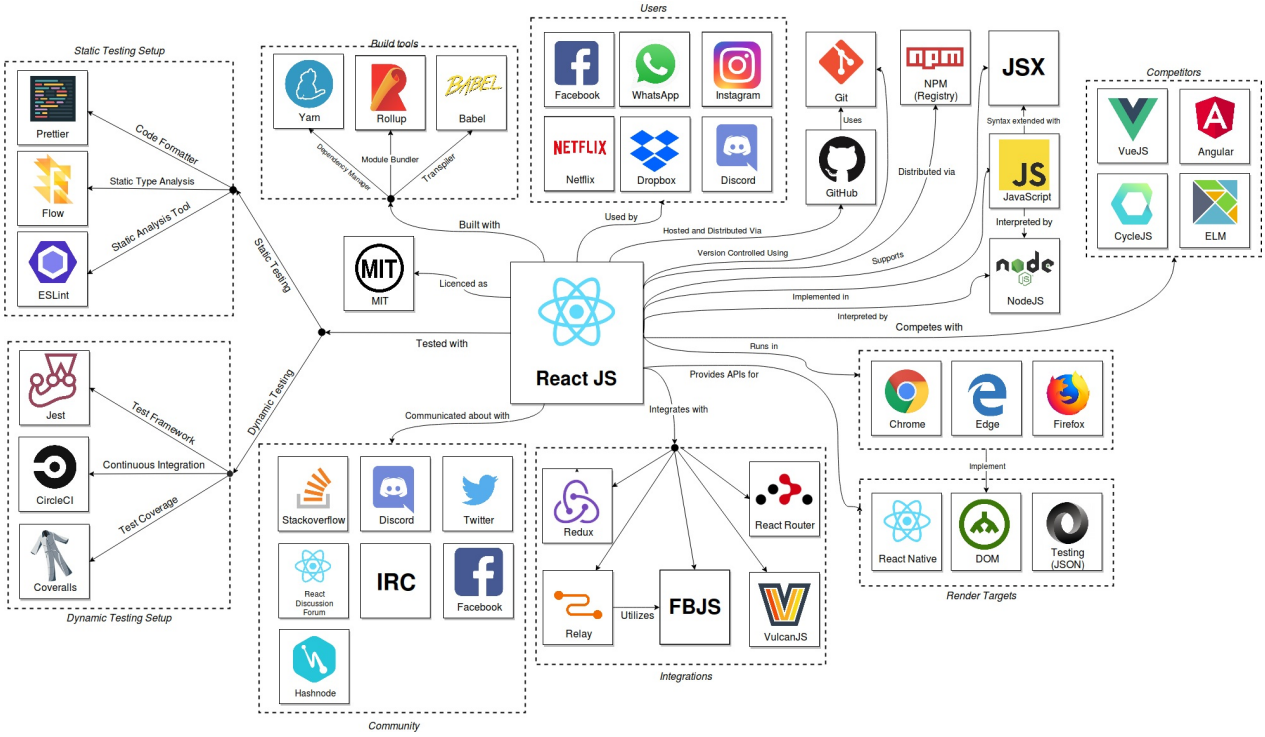


Figure 3.1: an overview of the external dependencies related to React.

The center of the diagram shows *React* itself surrounded by the many different external entities. The left side shows the grouped testing tools and they are further divided into two main testing setups: a static and dynamic testing setup. Next to the testing section on the left bottom the community is located which is a large group including seven entities. The right of the community shows the parties that integrate with React: the most important and popular entities have been picked and grouped here. Moving in counter-clockwise direction the renderers along with the browsers above them which implement the DOM are showed. In the right top corner the most popular and important competitors are visualized.

Starting in the left top corner again the build tools that *React* uses to build the codebase are displayed. The distribution tools that are used along with these building tools are on in the right top corner next to the competitors. *React* has a lot of users and next to the building tools the most popular software products that use *React* are grouped. Next to the users we see the version control services that the *React* contributors use to control their code.

3.3 External Entities

3.3.1 Testing tools

To ensure that developers can test *React* properly, a testing environment exists which contains static and dynamic testing tools. With ***Jest*** the *JavaScript* code is dynamically tested in the form of test suites. Then, through the utilization of *Yarn*, all the *Jest* test suites can be executed by the developers. Additionally, ***Coveralls*** will provide an analysis of the code and unit coverage that these tests produce. ***CircleCI*** is used to ensure that all *Jest* tests pass, which is done through the means of continuous integration (CI). This process ensures that development can be automated, which in turn provides an automated verification of correctness before code is deployed.

Besides *Jest* test files, the CI pipeline also runs the static analysis tools ***Flow***, ***ESLint*** and ***Prettier***. *Flow* is a static type checker for *JavaScript* and uses type inference to track data in the code that might lead to a bug. *ESLint* is a more general static analysis tool and is an open source project with the goal of providing a pluggable linting utility for *JavaScript*. *ESLint* will check for programming errors, bugs, stylistic errors, and other miscellaneous static errors. Finally, *Prettier* enforces that code is formatted in a correct and 'pretty' way, as defined by the style guide agreed upon by the *React* developers.

3.3.2 Version control services

React is an open source project which uses ***Github*** as its host. On *Github* an overview of the *React* codebase is shown. *Github* utilizes ***Git*** to function as version control of the codebase. Developers can employ *Git* to clone the codebase and maintain their own changes and those from others. Whenever a developer wishes to contribute, one must create a pull request on *Github* which will present the changes made to the codebase from their own Git branch to the core developers, whom can review and suggest necessary changes in cases of disagreement.

3.3.3 Building and distribution tools

JavaScript The library is implemented in the *JavaScript* language and should therefore be interpreted by a *JavaScript* runtime. In addition, various scripts related to building and testing the project are implemented that should be executed in the *NodeJS JavaScript* runtime.

Babel Relatively new syntactic *JavaScript* features (ECMA Script 2015+) are being used in the codebase of *React*. *Babel* is employed to transpile the code to an older syntax that is more broadly supported by adopted web browser versions. *Babel* interfaces with the project via a `.babelrc` configuration file in *JSON* format.

Rollup The *React* codebase consists of many modules, which are contained in separate *JavaScript* files. For *JavaScript* modules to be treated in a uniform way (such as by package managers), they must conform to a module format such as the *CommonJS* or *Asynchronous Module Definitions*. However, the ECMAScript 6 (ES6) revision proposes a native module system for importing and exporting functions and data between *JavaScript* files. Since challenges exist involving the implementation of this feature in *JavaScript* runtimes, *Rollup* is utilized which implements the functionality in the form of a package, thus allowing ES6 modules to be used. In addition, this allows *CommonJS* modules to be imported as well, ensuring a degree of backwards compatibility. The modules that compose the *React* codebase are managed and compiled in the form of ES6 modules by *Rollup*.

NodeJS Tasks for manipulating the *React* codebase are run in *NodeJS*, a *JavaScript* runtime based on Google's V8 Engine.

npm *npm* is the package ecosystem (registry, manager, and other features) that came with *NodeJS*. With over 350,000 packages (2016) *npm* is the world's largest open source software package ecosystem [51]. Furthermore, *npm* has had a negligible downtime. *React* is distributed as a package via the *npm* registry. It adheres to the CommonJS module format and manifests this by means of the `package.json` file.

Yarn Instead of the default *npm* package manager that comes bundled with *NodeJS*, *React* uses the *Yarn* package manager which is - coincidentally - a project operated by *Facebook*. Dependency *JavaScript* packages are retrieved from *Yarn*'s own package registry. *Yarn* interfaces with the project via a number of configuration files (`package.json` , `.yarnrc` , `yarn.lock`). The main file is a *YAML* file named `package.json` , describing *React* as a package itself as well as all its dependency packages.

3.3.4 Render Targets

React supports rendering the internally constructed UI component representation to different target formats / systems, handled by dedicated internal packages providing an interface to the targets. These targets consist of the following elements:

***DOM* (Browser)** *React* builds a virtual document object model (DOM) internally when constructing components. It then allows the rendering of these components to the actual DOM in a differential manner. This is achieved by only applying changes to those components in the DOM that differ from the corresponding component in the virtual DOM. This allows components to be efficiently rendered on web pages rendered by browsers.

React Native *React Native* is *Facebook*'s framework for mobile application development, relying on *React* for UI building. *React* contains a renderer that is meant as an interface for *React Native* specifically. The code in this thin layer, which provides a dedicated interface to *React Native*, is part of the *React* codebase (as opposed to the one of React Native) due to its high coupling with the *React* core.

***JSON* (Testing)** For testing purposes, the internal component representation can also be rendered to JavaScript Object Notation (*JSON*). This is utilized by the *Jest* testing framework for snapshot testing.

3.3.5 Integrating libraries and frameworks

The developers at Facebook maintain a utility library named **FBJS**, which consists of various in-house JavaScript packages, both to share these and to prevent duplication of code and effort. This library is a dependency for *React*, as well as some of its integrating packages.

Various libraries and frameworks integrate with *React*, and are often found to be used in combination. An example of this is the ***Redux*** state container, which is used to store and manipulate the state of an application. It has official bindings for *React*, such as smart components, which may listen to the store's state changes and presentational components get re-rendered when necessary. In addition, ***Relay*** is a framework that couples *React* and ***GraphQL*** - a query language and server-side runtime for evaluating them. This allows for a workflow for the creation of data driven *React* applications. Similarly, ***VulcanJS*** is an example of a full-stack web application that incorporates *React* for the UI building aspects. Finally, ***React Router*** is a set of navigational components allowing routing within a *React* application dynamically (i.e. as opposed to conventionally configuring routes up front).

3.3.6 Competitors

VueJS is a highly similar *JavaScript* library for building web user interfaces. It employs similar techniques of virtualizing the DOM. *React* and *VueJS* are regarded to be similar to such an extent that there have even been efforts to transpile components between the two libraries, such as with the Vueact library. Another *JavaScript* library with use cases similar to *React* is ***CycleJS***, which attempts to distinguish itself by putting an emphasis on including the human user more in the loop in order to improve the human-computer interaction aspects of the web interface. Contrary to these libraries, ***AngularJS*** provides an entire framework for building front-end web applications. Similarly to *React* it abstracts away from DOM manipulation to better facilitate the development of interactive web interfaces. ***Elm*** is a functional language that transpiles to *JavaScript*, having the specific purpose of programming interactive web applications.

3.3.7 Utilizing applications

Ultimately, the *React* library has the purpose of being incorporated into web applications to provide for and interactive front-end web interface. Numerous applications have done so, including *Netflix*, *Dropbox*, *Discord*, and *Facebook Inc.*'s internally developed applications such as *Facebook*, *WhatsApp*, and *Instagram*.

3.3.8 Communication channels

The *React* community is widespread across six applications. The ***Facebook* page** and the ***Twitter* account** are the main broadcasting platforms where news and articles related to *React* can be found.

To learn more about problems in *React* or to ask personal questions, one can use the platform ***Stack Overflow*** platform. This platform is known for providing a large question-and-answer base for a large amount of different languages, including *React*. The major developers at *React* ask to post code-level question on *Stack Overflow* while long-form discussions should be kept on the *React* Discussion Forum.

For a small question, one can go to *React*'s *Discord* page, which is called ***Reactiflux***, to quickly find developers who are online. In addition, some of the developers are also found on the ***IRC channel* #reactjs**.

React also uses ***Hashnode*** as a question-and-answer forum for discussions among developers.

4. Development View

The development view section aims to outline the module structure, codeline standards, and common design models of the *React* project.

4.1 Module Structure

A project's module structure represents the structural organization of the project’s source code into various inter-dependent internal modules.

4.1.1 Test Fixtures

Test fixtures automatically set up specific environments for testing purposes, so that a developer may quickly check in what ways, if any, their modifications affect some of *React*’s basic functionalities. The test fixtures for *React* consist of: *art*, *attribute-behavior*, *dom*, *expiration*, *fiber-debugger*, *fiber-triangle*, *packaging*, and *ssr*.

4.1.2 Scripts

Scripts are pieces of code that may be run to quickly and automatically achieve various goals. Developers can use scripts to run benchmarks, build and publish versions of *React*, and to lint files. The scripts for *React* consist of: *babel*, *bench*, *circleCI*, *error-codes*, *eslint-rules*, *eslint*, *facts-tracker*, *flow*, *git*, *jest*, *perf-counters*, *prettier*, *print-warnings*, release, *rollup*, shared, and tasks.

4.1.3 Core packages

Core packages provide the main functionalities of *React*. The core packages of *React* consist of: events, react-art, react-call-return, react-dom, react-is, react-native-renderer, react-noop-renderer, react-reconciler, react-test-renderer, react, shared, and simple-cache-provider.

4.1.4 Module Structure Model

Figure 4.1 represents the organization of *React*, and displays these as interdependent modules.

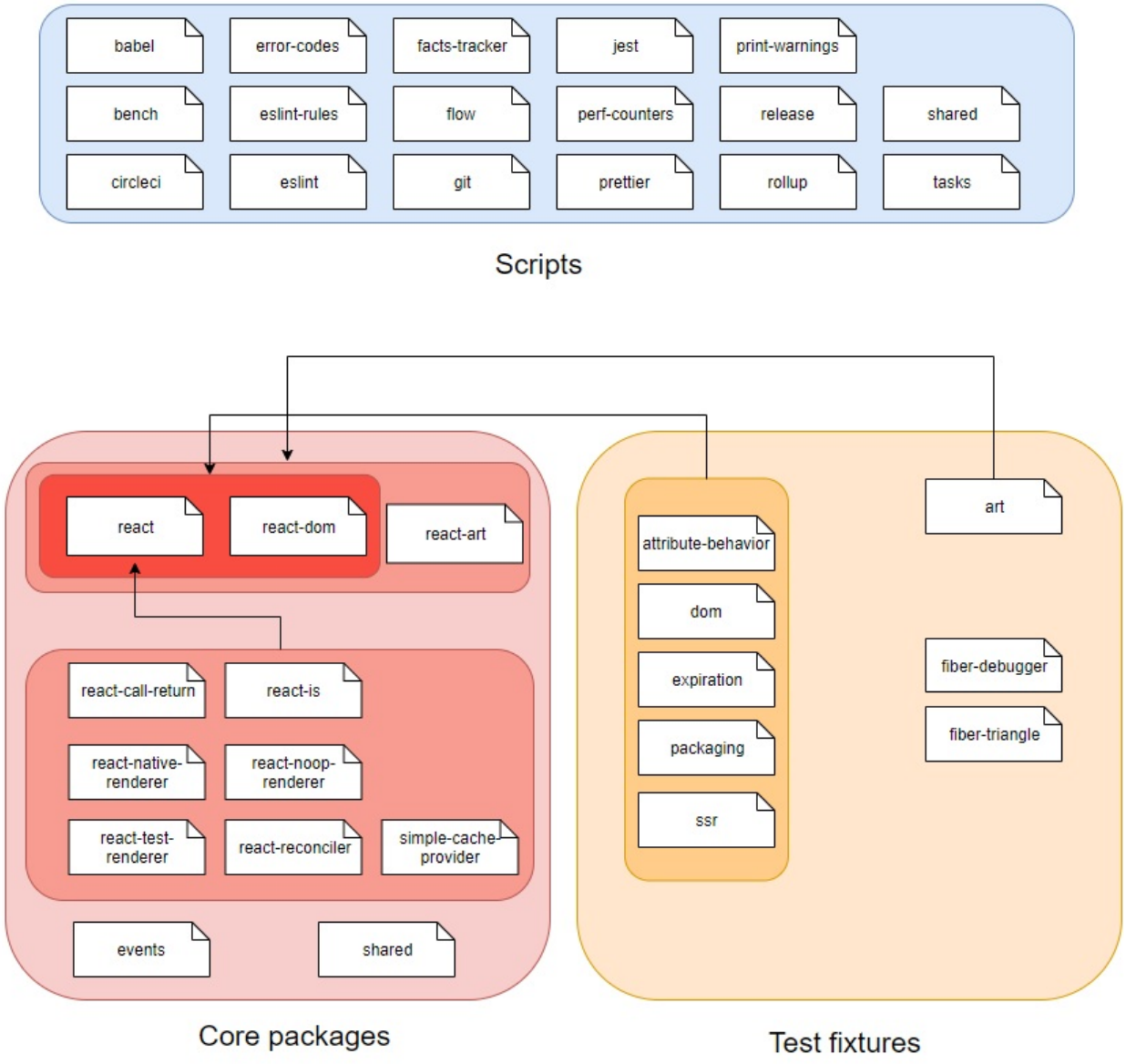


Figure 4.1: an overview of all the modules within React.

4.2 Codeline Model

The codeline model describes the organization of *React*'s codeline. This defines the structure, code control, different types of code, how the code should be maintained, and which automated tools are used to build, deploy, release, and test the code.

The codebase of *React* is located on the *Github* platform and is open source. This allows developers to clone the code to their own computer in an efficient and quick manner. The code on *Github* can be automatically built and tested through continuous integration (CI). For this, *CircleCI* is used which is a free tool to setup a CI pipeline. After developers make a push to the code base in a PR, their code will automatically be built and tested to prevent any unwanted errors, bugs, or formatting issues.

To deploy *React*'s code, one must be an *npm*-owner to upload the code to the *npm* registry. A few scripts have been setup to automate a large part of the process, which is all about making sure that the latest master branch build succeeds as well as all the test suites. After these important factors, the *npm*-owner who wishes to publish will also have to do some sanity checks such as ensuring that users will get the right version of dependencies at run time. It is also advised to run all the `yarn` components: `yarn test`, `yarn lint`, and `yarn flow`. When all process steps have been finished by the *npm*-owner, the release process can begin by running the build.js and publish.js script. The publish script will publish all build artifacts to *npm* and push it to *GitHub*.

For the developers to work concurrently they use *git* which provides version control for the codebase. For contributors it is necessary to first fork *React*'s repository so that they can make changes in any form they want. When changes are made to the fork they can be proposed to the React main repository by means of a pull request. Communicators or other in-house stakeholders will then review the changes made and accept or refuse the request.

Rollup addresses the desired structure of the code. Developers achieve this by installing the required plugins for *Rollup*. This is only required for the production build as using *Rollup* in the development build will also remove development warnings. The source code should be structured into a few packages as described in the figure 4.2.

Package	Description
events	*React* implements an event system which does not depend on the renderers and work with React DOM and React Native.
react-art	React-Art is used for drawing vectors, providing declarative and reactive bindings to sebmarkage's art-library, available at: https://github.com/sebmarkbage/art/
react-call-return	An experimental package for multi-pass rendering in *React*.
react-dom	This is all about the React DOM renderer. Can be used as a standalone browser bundle.
react-is	In this package one can set arbitrary test values and evaluate different objects of types specific to *React*.
react-native-renderer	Renders *React-Native* components to native views used in mobile applications.
react-reconciler	The different render packages share some code between them. This (unstable) package solves this issue of having duplicate code.
react-test-renderer	Renders *React* components to JSON trees. Mainly used for snapshot testing with *Jest*.
react	The core React code is located here. It includes the APIs necessary to define components.
shared	(We spent some time researching what this package is for, but were unable to find any information on it.)
simple-cache-provider	Unstable package which includes a cache for *React* applications.

Figure 4.2: Codeline package structure and relations.

4.2.1 Module inter-relatedness

A number of observations of the dependencies between modules may be gleamed from figure 4.1.

- Given that 'react' contains the most important code of the entire *React* project, it is no surprise that many of the other packages depend on it.
- There are no other dependencies amongst core packages; each package does its own thing, and at most needs only the 'react' core.
- A number of test fixtures depend on both 'react' and 'react-dom', which is "intended to be paired with the isomorphic react" [60]. Together, they may provide a complete core environment for test fixtures to build upon.
- The test fixture 'art' depends not only on 'react' and 'react-dom', but furthermore also on the 'art' core package.

4.3 Common Design Models

Common design models describe standardization agreements which dictate how developers and contributors must contribute to ensure that the *React* project remains of a high quality and well tested status. The major segments within the common design models for *React* are the standardization of code style, dependencies, and testing.

4.3.1 Standardization of Code Style

To ensure that all submitted code looks the same (identical indentation, variable naming, and other code style preferences), there are agreements regarding code style that must be followed within each contribution to *React*. Any contribution which does not adhere to the agreed upon code style standards will not be considered for implementation into the core release, as per the *React* contribution guide. The standardization of the code base consists of the following requirements:

- Code must be automatically formatted through the utilization of *Prettier* [52] and *Yarn* [53]. When running 'yarn prettier', all changes made to the code will be automatically formatted to the desired and agreed upon standards. To find remaining style guide issues remaining, a developer can then run 'yarn lint' to run the *linter* [61] for the project.
- All cases which are disputed and not resolved through the aforementioned tools will result in a default to match the *AirBnB Javascript* style guide [54].
- As described in the codeline model section, active enforcement of code style standardization takes place through, *CircleCI*, among others.

4.3.2 Standardization of Dependencies

React has a multitude of dependencies, which all require active monitoring to ensure the project remains stable. To avert this issue, all dependencies which *React* requires to function have become part of the *Facebook Inc.* body of repositories. This ensures that both legal and code stability requires are entirely covered and not dependent on external parties.

To further ensure the standardization of where dependencies are found, *React* requires the utilization of a package manager. *React* employs *Yarn*[53] for test and code formatting dependency deployment, and *npm*[56] for other dependency managements. By ensuring that all developers obtain their dependencies from the same package managers, it is guaranteed that all dependencies are identical, meaning compatibility issues don't propagate.

5. Technical Debt

This section contains all information regarding the technical status of *React*, the identification of technical debt, its evolution, and the discussions that took place regarding technical debt. A consideration is given to both the implementation itself, and also to technical debt related to testing.

5.1 Identification of Technical Debt

The identification of the technical debt within *React* was done through a combination of automated code quality analysis with the use of tools, and a manual code analysis taking into consideration the *SOLID* principles [62] for software design.

5.1.1 Automated Code Quality Analysis

Automated Code Quality Analysis refers to the utilization of tools that allow automatic validation of agreed upon static and dynamic code regulations. The static analysis refers to adherence to agreed spacing and code style requirements, whereas dynamic refers to code execution evaluation and the coverage provided by automated tests. These metrics provide an insight into the technical state of *React* and can assist in finding areas within the project that contain significant technical debt. These indebted areas, which are places in the code that require attention to fix technical issues, can then be selected for refactoring.

5.1.1.1 Static Code Analysis

The static code analysis aimed to identify issues within *React* that are related to code quality and code formatting. *React* relies on multiple tools to perform static code analysis, two of which are *Flow* [63] and *TypeScript* [64]. Although these two perform a similar function regarding code analysis, *React* implements both and has *JSON* files and configurations that separately perform similar analysis over similar components. This is a form of technical debt, as redundancy usually indicates a disagreement or refactoring took place, and there was no decision made regarding which implementation should be kept.

Beyond this duplication, running the static code analysis as described in the *React* contribution guide [65], no static code errors exist in the master branch of the code. This is due to the requirements outlined in the contribution guide, which dictate that any pull requests or submissions to the codebase with these errors will be rejected on those grounds, ensuring a low technical debt and high static code quality.

5.1.1.2 Dynamic Code Analysis

The dynamic code analysis was performed by measuring the test coverage generated by the latest release of the master branch, as reported by the in-house *Coveralls* coverage tool [66]. The decision was made to only analyze the core, *create-subscription*, *dom*, and * *folders to avoid accidentally considering* React Native *or testing packages. Packages meant for distribution or non-functional purposes, such as files aimed at the* npm* package manager, are also ignored. The statistics for each package can be found table 5.1.

Package	File / Folder	Coverage %
Create-Subscription	createSubscription.js	88.04%
Core	React.js	86.21%
Core	ReactBaseClasses.js	88.24%
Core	ReactChildren.js	89.85%
Core	ReactContext.js	86.36%
Core	ReactCreateRef.js	85.71%
Core	ReactCurrentOwner.js	100.00%
Core	ReactDebugCurrentFrame.js	92.86%
Core	ReactElement.js	85.93%
Core	ReactElementValidator.js	90.91%
Core	ReactNoopUpdateQueue.js	82.14%
Core	forwardRef.js	83.33%
DOM	client	94.79%
DOM	events	88.56%
DOM	server	95.61%
DOM	shared	96.35%
DOM	test-utils	86.67%
DOM	unstable-dependencies	100.00%
noop-renderer	ReactNoop	64.90%
	TOTAL	89.92%

Table 5.1: overview of packages/files, and their associated test coverage percentage value.

As the table shows, the overall coverage for *React* is 89.92%, a significant coverage metric, indicating good code quality. However, the largest discrepancies in coverage results are found in the *reactCurrentOwner*, *unstable-dependencies*, and *ReactNoop*. The *reactCurrentOwner* and *unstable-dependencies* sections are both positive outliers due to their perfect test coverage. This score is attributed to the fact that these classes validate that certain subclasses or dependencies are called in a certain order. If any tests invokes any part of these sub-classes, the top level classes receive maximum possible coverage. The negative case is the *ReactNoop JavaScript* file, which is a renderer aimed at allowing for the testing of semantics outside of the actual *React* environment. This class has a low coverage, mostly due to the fact that branches are barely tested and error handling is not tested. Since this class aims to validate semantic behavior, many catch-throw operations take place, yielding many errors, which are not tested.

Due to the finding of a lack of error catching testing, a further manual analysis was done based on the code coverage report. Analysis shows that many instances of test coverage decreases can be attributed to a lack of error handling testing. Instances in which code equivalent to that found in the figure below is located are rarely tested, significantly decreasing code and branch coverage for *React*.

```
if CONDITION TO TEST < SHOULD NOT OCCUR {
  throw new Error('This is the error message of an untested error.');
}
```

Figure 5.2: theoretical example of untested failure cases.

Of the packages that have less than 80% coverage, *react-call-return* and *simple-cache-provider* are noted in their respective README's to be very unstable and prone to changes, which explains their lower test coverage. Furthermore, *react-noop-renderer* is not meant for direct use. *React-art*, which has the lowest result of these packages, is used primarily for drawing. It is speculated that, since a flaw in this package would only cause cosmetic errors, thorough testing was deemed of lesser importance for this particular package, but this is mere conjecture.

5.1.2 SOLID Analysis

The *SOLID* principles [62] consist of ideas that a project must adhere to with regards to class responsibilities and interaction to ensure the complexity remains at an acceptable level while also being maintainable. The principles are closely related to the object-oriented programming (*OOP*) principles. *SOLID* consists of the **Single Responsibility Principle**, **Open-Closed Principle**, **Liskov Substitution Principle**, **Interface Segregation Principle**, and the **Dependency Inversion Principle**. *React* was analyzed based on these principles, and the results for each principle can be found below in the respective sub-category. Class identification was performed in part based on the findings from the *static* and *dynamic code analysis* sections.

5.1.2.1 Single Responsibility Principle

Although *React* does a good job attempting to organize all the classes and packages based on their functionality (*DOM*, *noop-renderer*, core), there are still classes which violate this principle. The clearest violation of the SRP is the *ReactDOM* class [67], found in the *DOM* package. This class, clocking in at 1354 lines of code (LOC), is not only responsible for the updating and handling of the DOM elements, but also for managing related dependencies such as batching and event registration. This violation of the SRP is so abundant that even the developers have made a *TODO* comment regarding this behavior to fix it. This comment, including the importing of responsibility classes, is shown in figure 5.3.

```
import '../shared/checkReact';
import './ReactDOMClientInjection';
import ReactFiberReconciler from 'react-reconciler';
// TODO: direct imports like some-package/src/* are bad. Fix me.
import * as ReactPortal from 'shared/ReactPortal';
import ExecutionEnvironment from 'fbjs/lib/ExecutionEnvironment';
import * as ReactGenericBatching from 'events/ReactGenericBatching';
import * as ReactControlledComponent from 'events/ReactControlledComponent';
import * as EventPluginHub from 'events/EventPluginHub';
import * as EventPluginRegistry from 'events/EventPluginRegistry';
import * as EventPropagators from 'events/EventPropagators';
import * as ReactInstanceMap from 'shared/ReactInstanceMap';
import {enableCreateRoot} from 'shared/ReactFeatureFlags';
import ReactVersion from 'shared/ReactVersion';
import * as ReactDOMFrameScheduling from 'shared/ReactDOMFrameScheduling';
import {ReactCurrentOwner} from 'shared/ReactGlobalSharedState';
import getComponentName from 'shared/getComponentName';
import invariant from 'fbjs/lib/invariant';
import lowPriorityWarning from 'shared/lowPriorityWarning';
import warning from 'fbjs/lib/warning';
import * as ReactDOMComponentTree from './ReactDOMComponentTree';
import * as ReactDOMFiberComponent from './ReactDOMFiberComponent';
import * as ReactInputSelection from './ReactInputSelection';
import setTextContent from './setTextContent';
import validateDOMNesting from './validateDOMNesting';
import * as ReactBrowserEventEmitter from '../events/ReactBrowserEventEmitter';
import * as ReactDOMEventListener from '../events/ReactDOMEventListener';
import {getChildNamespace} from '../shared/DOMNamespaces';
```

Figure 5.3: violation of the SRP in react-DOM.

A violation of the SRP commonly indicates that too many responsibilities were added as development continued without a clear managing or central entity which manages this. This, in turn, can show signs of a lack of planning or refactoring, and thus the accumulation of technical debt. These classes could be refactored through central discussions of lead developers, who could agree on a new standard and refactor this.

5.1.2.2 Open-Closed Principle

React clearly violates this principle through the implementation of the renderer. React utilizes the *noop* renderer to validate behavior at a more syntactical and abstract level, which allows for easier testing of implemented logic. The *noop* renderer is stored in a separate package with a single source file [69], whereas the updater hooks for *noop* are placed in the *ReactNoopUpdateQueue* class in the root package [70].

If *React* were to adhere to the OCP, an attempt would be made to extend the update queue class based on the native *noop* update queue, or the *noop* renderer found in *React* would be extended to only re-implement the differentiated features of the native *noop* renderer. This behavior and violation of the OCP exists in multiple other locations within *React*, and commonly indicates a lack of developer willingness to utilize the native source dependencies to create desired functionality. This, in turn, means that the tested functionality found in these dependencies can possibly be broken. This, in turn, creates additional technical debt due to tests that can start failing as a result of the OCP violating extensions.

It should be noted that *React* itself is easily extendable, thereby adhering to the OCP. *Facebook* has done this themselves with *React-DevTools* [72], a project which provides developers additional insights into *React* by extending the core of the project.

5.1.2.3 Liskov Substitution Principle

Within *React*, this behavior occurs in multiple instances, but the clearest violation is, again, the *noop* renderer. When attempting to utilize the renderer as an extension to the *noop* class in the source package, multiple violations will occur. This principle is important to track, since violations of it commonly indicate that a subclass is too dependent on changing behavior of its associated super class, meaning single change in this super class can break the subclass. This would then result in the increase of errors, and hence, the overall technical debt.

5.1.2.4 Interface Segregation Principle

If clients were too dependent, then changing hooks would result in the required updating of components within *React* or by partners, thus decreasing the ease of implementation and thereby increasing technical debt. *React* clearly segments its packages, as shown in table 5.4.

Package	Purpose
create-subscription	Allows subscriptions to external, non-*React* data.
events	Ensures changes required to be caught can be caught, separate from *React* core.
react-ART	Allows for creation of vector graphics in *React*.
react-call-return	Experimental implementation of multi-pass rendering.
react-dom	*React*'s implementation of the domain object model.
react-is	Package for testing of values to see if they are a *React* type.
react-native-renderer	The renderer used for the *JavaScript*-less version of *React*.
react-noop-renderer	The renderer used for the *noop* purposes.
react-reconciler	Allows the creation of custom *React* renderers.
react-test-renderer	Renders *React* components without reliance on the *React* DOM.
react	Core folder, provides the library.
shared	Contains classes and functionality required by all elements.
simple-cache-provider	Basic cache for *React*.

Table 5.4: described purposes of React interfaces.

Due to the segregation, developers can clearly see which elements they must edit, and all shared responsibility can be found in a single location. Overall, the *React* implementation indicates that experimental and additional library functionality is separated from the core functionality. This decreases overall technical debt, as specific features can be isolated for implementation, and shared elements are tested uniformly and reviewed by all developers.

5.1.2.5 Dependency Inversion Principle

As shown in the interface segregation section, *React* segregates based on functionality, but also within the project itself. As the package layout shows, an effort was made to split renderers based on functionality, and dependencies based on external hook requirements (a developer implementing *React* desires the event package, but not the core package). Due to this, bugs can be avoided, as changing modules does not cause a change in functionality throughout the system nor its implementing modules.

5.3 Evolution of Technical Debt

Originally, React was developed internally at Facebook by Jordan Walke. The initial public version was released in May 2013. Over the course of the past 5 years, React went through numerous architectural changes. This section describes a few specific cases of major changes to the design and architecture of React that had a major impact on its technical debt. We first analyze the re-implementation of the core algorithm known as *reconciliation* over the course of 2016 and 2017 as part of the release of React 16.

The evolutions described in this chapter were identified by elaborately inspecting the React documentation, the repository revision history, GitHub Issues and Pull-Requests, meeting notes and conference talks. In a later iteration, we plan to identify additional cases by using automated tools on the codebase for example to generate and analyze an Evolution Matrix. In addition we aim to describe a number of other smaller evolutions that were identified in the documentation.

5.3.1 Reconciler Re-implementation

In this section, we explain the 'reconciler', which is considered to implement the core algorithm of React. We then describe how this subsystem has evolved over time by addressing a major re-implementation. Finally we address how this evolution relates to technical debt.

5.3.1.1 Purpose of the Reconciler

React provides a declarative API to developers for declaring components of a user interface (UI). This means that determining which actions to perform in order to reflect changes in the state of the UI is entirely handled by the library. This is referred to as a *pull* approach as opposed to *push* approach. The naive approach to implement this behavior would be to completely rebuild the UI on each state change. This might work in small trivial applications, but it quickly becomes expensive in terms of performance for larger applications.

To solve this problem, React employs an algorithm for determining a minimal amount of changes to the UI necessary to reflect an updated state. Initially, a representation of the UI is rendered in the form of a tree of nodes, which is internally cached. This tree is then flushed to the render target (e.g. the DOM) in order to effectuate it. Any subsequent changes to the state will result in the tree being re-rendered. The resulting tree is then compared to the cached tree using a diffing algorithm to determine the minimal amount of changes necessary to transform the cached tree into the new tree. Only these changes are then flushed to the render target. As a benefit, the reconciler can be reused in combination with different renderers, since the changes are described in a render-target-agnostic way.

The problem of comparing trees this way in general can be solved using an $O(n^3)$ algorithm, but this is not fast enough in practice. React implements a diffing algorithm named 'reconciliation' that uses two main heuristics to improve performance as listed in the Reconciliation documentation:

> - Two elements of different types will produce different trees.
> - The developer can hint at which child elements may be stable across different renders with a key prop.

5.3.1.2 Evolution process

After a couple of years of research on the initial *Stack Reconciler*, the React team decided to completely overhaul the core reconciliation algorithm. Right after the release of React 15, the target of re-implementation of the reconciler was set for the next milestone of the React 16 release, as documented in the meeting notes of April 4th, 2016. Sebastian Markbåge (@sebmarkbage) appears to be the lead developer of this new reconciler. Initially, the new reconciler is referred to as the *incremental reconciler*, although later it would be named the *Fiber Reconciler*. In the notes of April 21th some first concerns with regards to the re-implementation are referred to, as expressed and discussed upon earlier in Issue 6170 (March 2nd). The first commit for the new reconciler was merged as part of PR 6690. Later meetings of June 9th, June 23rd, August 4th, October 13th, October 20th, November 3rd and December 1st each record updates on the development of Fiber. In order to allow other developers to start contributing to the Fiber Reconciler, Sebastian published the principles behind the new reconciler in Issue 7942 on October 11th. The last recorded core team meeting notes of December 8th record that Fiber has been successfully employed in various example applications, and that is set to rollout early 2017. The completion status of Fiber - which had gained quite some public attention being a major new feature - was tracked publicly on a dedicated website. There were some concerns about backwards compatibility as discussed in Issue 9463 starting from April 20th, 2017, but the core developers guaranteed it would work out of the box due to their focus on feature parity. The release of v16.0.0 on September 16, 2017 - later than anticipated - was the first version to include the new Fiber reconciler, as reported in the corresponding blog post.

5.3.1.3 Paying the Technical Debt

The re-implementation had the general purpose of paying technical debt in numerous ways. The re-implementation allowed the team to solve various long standing issues like readability and anti-patterns in what seems to be the most complex part of the codebase in one fell swoop. This is for example expressed in the meeting notes of October 13th, 2016:

> It seems like Fiber is our best shot at fixing many long-standing issues with React, and we are going to place all our effort into either replacing existing reconciler with Fiber, or failing spectacularly with it (and learning from that).

However, these opportunities seemed to be side issues, as the re-implementation was mainly motivated by the problem that the synchronous uninterruptible rendering process often impacted user experience in large applications. As a solution, work like diffing and rendering would have to be scheduled and prioritized. This would allow for example rendering of animations to be prioritized over

fetching data from external sources. In addition, concurrency plays an important role: if the rendering / diffing process were to be split up into different interruptible units of work, low priority work could be paused for a higher priority workload, and resumed afterwards.

The old implementation of the reconciliation algorithm did not allow for these critical improvements, which can be identified as a large expansion of technical debt. Fiber paid this debt by implementing scheduling and concurrency as described above. More specifically, it implements a virtual call stack specific to the domain of building user interfaces. Each virtual stack frame (i.e., a unit of work) is referred to as a "fiber", hence the name of the reconciler. An analogy between Fiber and the more general concept of the call stack can be made as shown in table 5.5.

Call Stack	Fiber
Subroutine	Component Type
Body *(Nested subroutine invocations)*	Children
Return Address	Parent Component
Arguments	Props
Return Value	Tree Nodes *(render target specific output)*

Table 5.5: analogy between the call stack and Fiber.

The high-level architectural changes brought about by the introduction of Fiber are further elaborated upon by Andrew Clark (@acdlite) in his Fiber Architecture description and his Talk at ReactNext 2016 on Fiber.

6. Conclusion

Despite *React* being a large project with many contributors and large packages, it manages to maintain a high level of test coverage and an overall well organized codebase. Their codeline model and requirements for developers are clear and organized in a good fashion. Hence, we conclude that *React* is a project which is acting at potential and spending time at taking care of technical debt and issue within the project. This could be seen through their reconciler re-implementation. A significant effort is also put into keeping the codebase segregated based on features, thereby also making it easier to have an overview of the project and all possible permutation regarding renderers and other options.

Future focuses for *React*, based on this research, should consist of managing their documentation and continuing the current trend regarding code quality and organization.

Please note that a more extended version of this chapter, including additional information, is available at our chapter repository.

References, Footnotes, & Works Cited

[1] Nick Rozanski and Eoin Woods. 2012. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Chapter 9, pages 135-138. Addison-Wesley.

[2] Facebook Inc. Our History. Accessed at https://newsroom.fb.com/company-info/ on 22 February 2018.

[3] ReactJS, 2018. How to Contribute. Accessed at https://reactjs.org/docs/how-to-contribute.html on 22 February 2018.

[4] Github. React/License. Accessed at https://github.com/facebook/react/blob/master/LICENSE on 22 February 2018.

[5] Github. React/package.json. Accessed at https://github.com/facebook/react/blob/master/package.json on 23 February 2018.

[6] Nick Rozanski and Eoin Woods. 2012. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Chapter 9, page 135, Communicators. Addison-Wesley.

[7] https://github.com/gaearon

[8] https://github.com/aweary

[9] https://github.com/bvaughn

[10] https://stackoverflow.com/questions/tagged/reactjs

[11] https://discuss.reactjs.org

[12] https://twitter.com/reactjs

[13] https://facebook.com/react

[14] https://vuejs.org

[15] https://angular.io/

[16] https://elm-lang.org

[17] https://cycle.js.org

[18] ReactJS, 2018. How to Contribute. Accessed at https://reactjs.org/docs/how-to-contribute.html on 2 March 2018.

[19] https://github.com/sophiebits

[20] https://github.com/zpao

[21] https://github.com/sebmarkbag

[22] https://github.com/acdlite

[23] https://github.com/petehunt

[24] https://github.com/chenglou

[25] https://github.com/vjeux

[26] https://github.com/jimfb

[27] https://circleci.com/gh/facebook/react

[28] https://coveralls.io/github/facebook/react

[29] https://medium.com/netflix-techblog/netflix-likes-react-509675426db

[30] https://yahooeng.tumblr.com/post/101682875656/evolving-yahoo-mail

[31] https://www.youtube.com/watch?v=tUfgQtmG3R0

[32] https://blog.discordapp.com/using-react-native-one-year-later-91fd5e949933

[33] https://facebook.github.io/react-native/

[34] Wolf, Adam. *Explaining React's license*. Facebook. Accessed at https://code.facebook.com/posts/112130496157735/explaining-react-s-license/ on 13 March 2018.

[35] https://opensource.org/licenses/BSD-3-Clause

[36] https://code.facebook.com/pages/850928938376556

[37] https://www.apache.org/legal/resolved.html#category-x

[38] Mullenweg, Matt. *On React and WordPress*. Accessed at https://ma.tt/2017/09/on-react-and-wordpress/ on 13 March 2018.

[39] https://twitter.com/reactjs/status/911347634069168128

[40] Mullenweg, Matt. *Facebook Dropping Patent Clause*. Accessed at https://ma.tt/2017/09/facebook-dropping-patent-clause/ on 13 March 2018.

[41] https://github.com/facebook/react/commit/b765fb25ebc6e53bb8de2496d2828d9d01c2774b

[42] https://github.com/reactjs/core-notes

[43] https://preactjs.com/

[44] https://github.com/reactjs/core-notes

[45] https://stackoverflow.com/questions/tagged/reactjs

[46] https://discuss.reactjs.org/

[47] https://discord.gg/0ZcbPKXt5bZjGY5n

[48] http://irc.lc/freenode/reactjs

[49] https://github.com/markerikson/react-redux-links

[50] https://www.reactiflux.com/learning/

[51] https://www.linux.com/news/event/Nodejs/2016/state-union-npm

[52] - https://prettier.io

[53] - https://yarnpkg.com

[54] - https://github.com/airbnb/javascript

[55] - ReactJS, 2018. How to Contribute. *Style Guide*. Accessed at https://reactjs.org/docs/how-to-contribute.html on 2 March 2018.

[56] - https://www.npmjs.com/

[57] - https://facebook.github.io/jest/

[58] - ReactJS, 2018. How to Contribute. *Sending a Pull Request*. Accessed at https://reactjs.org/docs/how-to-contribute.html on 6 March 2018.

[59] - https://flow.org/

[60] - https://github.com/facebook/react/tree/master/packages/react-dom

[61] - https://eslint.org/

[62] Ramirez, Cristian. *S.O.L.I.D The first 5 principles of Object Oriented Design with JavaScript*. Accessed at https://medium.com/@cramirez92/s-o-l-i-d-the-first-5-priciples-of-object-oriented-design-with-javascript-790f6ac9b9fa on 13 March 2018.

[63] https://flow.org/

[64] https://www.typescriptlang.org/

[65] ReactJS, 2018. *How to Contribute*. Accessed at https://reactjs.org/docs/how-to-contribute.html on 15 March 2018.

[66] https://coveralls.io/builds/15992318

[67] https://github.com/facebook/react/blob/master/packages/react-dom/src/client/ReactDOM.js

[68] Meyer, Bertrand (1988). *Object-Oriented Software Construction*. Prentice Hall.

[69] https://github.com/facebook/react/blob/master/packages/react-noop-renderer/src/ReactNoop.js

[70] https://github.com/facebook/react/blob/master/packages/react/src/ReactNoopUpdateQueue.js

[71] Martin, Robert (2002). *Agile Software Development: Principles, Patterns and Practices*. Pearson Education.

[72] https://github.com/facebook/react-devtools

[73] https://github.com/markerikson/react-redux-links

[74] https://www.fullstackreact.com/30-days-of-react/day-22/

[75] https://www.robinwieruch.de/tips-to-learn-react-redux/#react-test-often

[76] http://reactkungfu.com/2015/07/approaches-to-testing-react-components-an-overview/

Apache Spark




@chialun-yeh @hrayo712 @vpourquie @arucard21

By Chia-Lun Yeh, Hiram Rayo Torres Rodríguez, Valérie Pourquié, and Riaas Mokiem.

Abstract

Apache Spark is a fast and general engine for large-scale data processing. Developed originally at the University of California, Berkeley's AMPlab by Matei Zaharia, and later donated to the Apache Software Foundation, which has maintained it since. Spark was designed specifically to increase performance on specific use-cases like machine learning where Hadoop's MapReduce performed poorly.

This chapter provides an insight of the Spark Core architecture by following the viewpoints and perspectives architectural description suggested by Nick Rozanski and Eoin Woods in their book. We give a description of the stakeholders and an insight of the context in which Spark is developed. Then, we proceed to describe the Functional view, the Performance perspective, and the Development view, to show how Spark was designed in order to meet its original requirements and how it is implemented. Also, an analysis of technical debt on the system is made, and finally, we conclude by giving our opinion on the previously mentioned aspects.

Introduction

Apache Spark is an open-source framework for processing large amounts of data. It is described as "a fast and general engine for large-scale data processing" [1], providing up to 100x faster performance than Hadoop MapReduce when the data can be read from memory and up to 10x faster performance when the data has to be read from disk. It is a Top-Level Project of the Apache Software Foundation [2] and is one of the most active projects in the Apache Software Foundation [3].

Apache Spark was created after specific use-cases, like machine learning, were shown to perform poorly using Hadoop MapReduce. Alleviating these problems provided the motivation for creating Spark [4].

After some analysis of the Apache Spark project, it became apparent that it actually contains many distinct products. The prominent ones that are singled out on the website are Spark SQL, Spark Streaming, MLib, and GraphX. These products provide specialized functionality on top of the main Spark product, Spark Core.

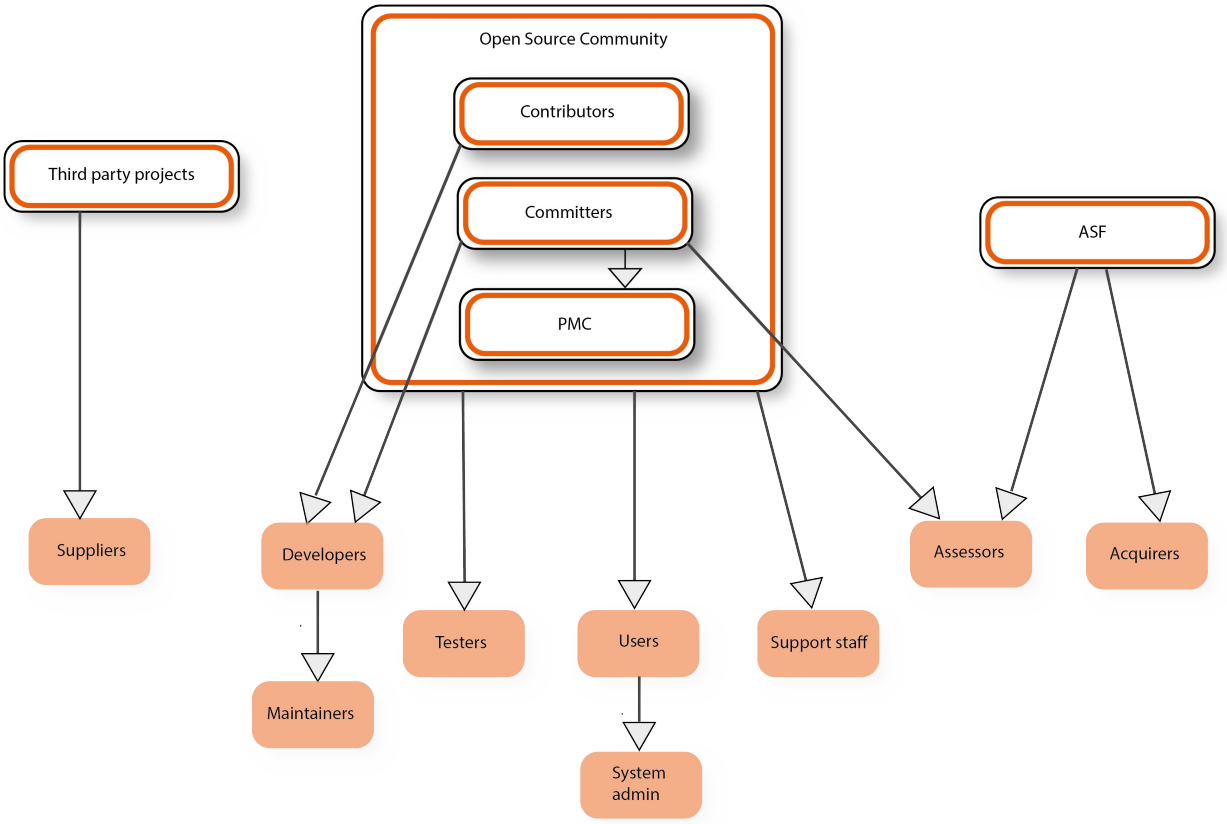
Given the complexity of the Apache Spark project and our time constraints, we decided to focus on the main product of the Apache Spark project, Spark Core. It provides the essential functionality used in Spark since all other Spark products are built around it.

In this chapter, we try to provide some insight into the architecture of Spark Core. We first describe who the stakeholders are in the Stakeholders Analysis, and how Spark Core fits into the world around it, in the Context view. We then describe the Functional view, the Performance perspective and the Development view, which should provide insight into what Spark Core does, how it was designed to meet its performance requirements and how it was implemented. We also show how technical debt is dealt with Spark Core and conclude with our opinion on the architecture of Spark Core.

Stakeholders

Apache Spark Stakeholders

For the Apache Spark project, we have identified the following stakeholders [6]:



1. Third-party projects
   - Being an open-source project, Apache Spark makes use of other open-source projects which can then be considered their suppliers. These include the libraries that they use to build the product, like Scala or Netty.
2. The open-source community
   - The open-source community is a common term used for the people that are gathered around a piece of open-source software, in this case, Apache Spark. The community consists of a variety of people, ranging from non-technical to technical, each of which can perform a range of tasks.
   - There are 2 specialized groups within the community.
     - *Contributors*: These people modify the source code of Apache Spark and submit Pull Requests to get these modifications into the main codebase. They are *Developers*.
     - *Committers*: These people also modify the source code of Apache Spark, so they are also *Developers*. They do many

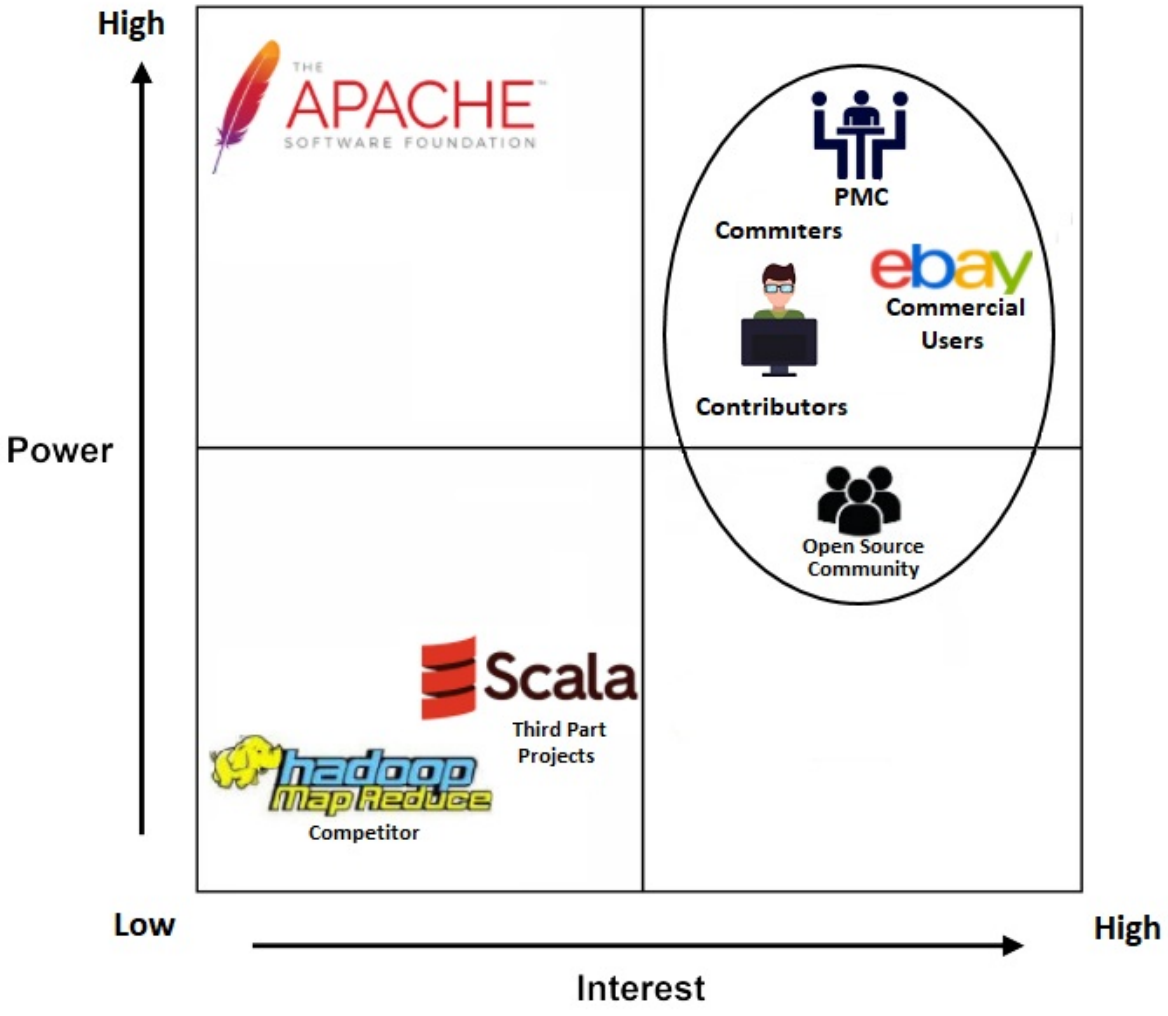
other things as well, performing as different stakeholders. As *Integrators* they accept and merge Pull Requests but they also safeguard the codebase as *Assessors*.

- The remainder of the community consists of companies, people in the community or educational users. These community members don't just use the software, they also contribute by providing feedback and supporting others. As stakeholders, this makes them *Users*, *Testers*, *Support Staff*, and *Communicators*.
- A subset of the standard *Users*, called *Commercial Users*, use the product as part of their core business. They would benefit from having power over Apache Spark so some measures are taken so they can't get too much power. For example, the ASF has a policy of only allowing participation of individuals and tries to ensure that a PMC does not consist of too many individuals from a single company [8].

3. The Apache Software Foundation (ASF)
   - As *Acquirer*, the ASF handles financial contributions and distributes it to their associated projects.
   - As *Assessor*, the ASF ensures the proper licensing of the project and creates and enforces several policies.
   - As *Facilitator*, the ASF provides and supports processes used by Apache Spark. They can use the funding provided as *Acquirer* more effectively by pooling the resources of multiple projects to facilitate these processes.
4. Hadoop MapReduce can be considered the main *Competitor* of Apache Spark. There are other *Competitors* like Apache Heron and Apache Storm though these are intended for stream processing.

Power and Interest

To change a system, a stakeholder needs to be interested in changing it and have the power to do so. We can map power and interest for each stakeholder to see how influential they can be, as shown in Figure 2.



We can see that for most stakeholders, interest and power increase together, though power more slowly. The *Committers* are high in both interest and power since they are the ones that maintain and shape Apache Spark. They decide what gets included and what doesn't. *Contributors* have less power since they can create code but it's up to the *Committers* to incorporate this in the system. *Commercial Users* have a similar degree of power since they can provide manpower and funding. This, indirectly, gives them quite a bit of power. This is useful to them due to their commercial use of the system, which gives them the highest interest of all stakeholders. The rest of the community can provide feedback and support which gives them less power, but they are just as interested as the stakeholders mentioned so far.

We then have some stakeholders with low interest and power. *Competitors* will likely have low interest in influencing the system as well and won't have much power to do so. *Third-party projects* that use Apache Spark will have a slightly higher amount of interest but won't have much power either.

A clear outlier in this figure is the ASF. Being both an *Acquirer* and *Facilitator* gives the most power to influence development. But its interest in doing so is very low. As *Facilitator*, its interest is just to support the project. It might have had more interest as *Acquirer* but the project is open-source and the ASF is a charitable organization so there's low interest financially. So while having the most power, the ASF has very low interest and is unlikely to influence the development of the system.

Context View

The Context view describes the relationships, dependencies, and interactions of a system with its environment as well as its scope and high-level requirements or responsibilities [5]. In this part of the chapter, we will discuss the context of Spark Core by explaining the system scope and analyzing the relationships of the system with all its external entities and interfaces.

System Scope and Responsibilities

Apache Spark began in 2009 as a research project at UC Berkeley AMPlab. At that time, Hadoop MapReduce was the dominant parallel programming engine for clusters, but it had big latencies because data was read and written serially for each job. Spark was thus initiated with the objective to perform in-memory cluster computing.

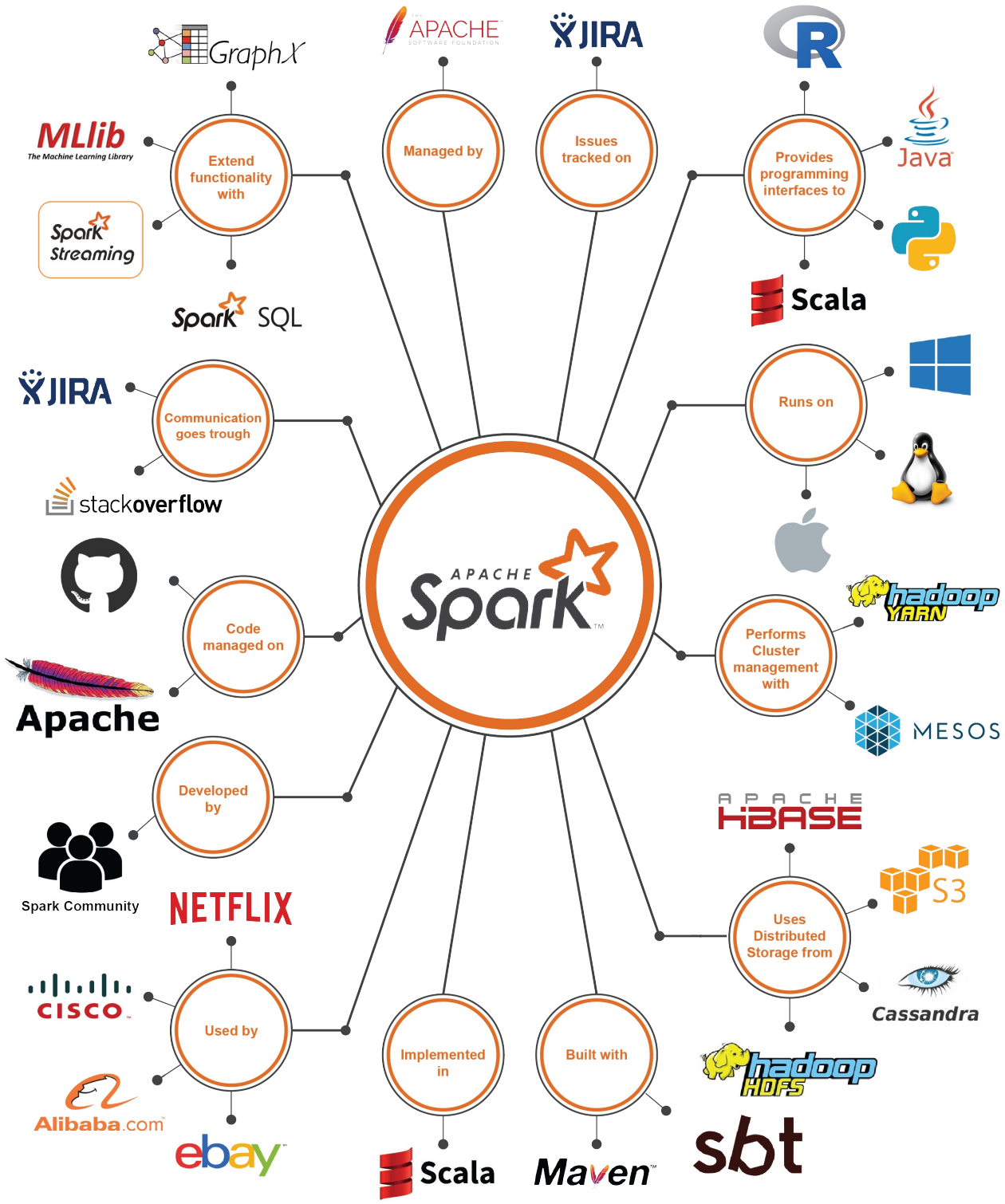
The responsibilities of Spark Core is to provide distributed task dispatching, scheduling, and basic I/O functionalities. It loads data from a distributed storage system and takes instructions from an application to decide how the data should be processed. It works with the cluster manager to distribute the data across the cluster so that data can be processed in parallel. Note that Spark does not provide permanent data storage, instead it makes use of already existing storage systems.

As stated in the first implementation of Spark [9], the system achieves the following:

> 1. It supports batch, interactive, iterative and streaming computations in the same runtime, enabling rich applications that combine these modes and offering significantly higher performance for these combined applications than disparate systems.
> 2. It provides fault and straggler tolerance across these computation modes at a very low cost.
> 3. It achieves performance that is often 100× better than MapReduce, and comparable with specialized systems in individual application domains.
> 4. It allows applications to scale up and down elastically and share resources in a responsive fashion.

External Entities and Interfaces

The environment in the Context view is defined as the external entities such as service providers, users, and competitors, as well as the interfaces to those entities. To better understand the system scope and analyze the external entities with which it interacts, we provide the context model of Spark as shown in Figure 3. Since we are only looking at Spark Core in this analysis, the additional libraries of the Spark project (Spark SQL, MLib, etc.) are considered external entities.



In the following we explain the context model in more detail:

- **Cluster Manager**: Spark can use different cluster managers and usually a third-party cluster manager will be used in production environments. Spark also contains a built-in cluster manager that works locally and non-distributed, but this is mostly intended for development and debugging purposes. Using a distributed cluster manager, all applications are run as independent processes on a cluster, all coordinated by a central manager that acts as the *service provider*.
  - The **external interface** for the cluster manager is provided through the use of *interfaces in the code*, most notably the *ExternalClusterManager* interface. These interfaces have to be implemented for each cluster manager. Currently, the following implementations are available: *Apache Mesos*, *Hadoop YARN*, and *Kubernetes*.
- **Storage System**: Spark requires a storage system that stores data permanently. To handle the big amount of data that a given application may require, Spark can interface with a wide variety of file systems, such as *Hadoop Distributed File System (HDFS)*,

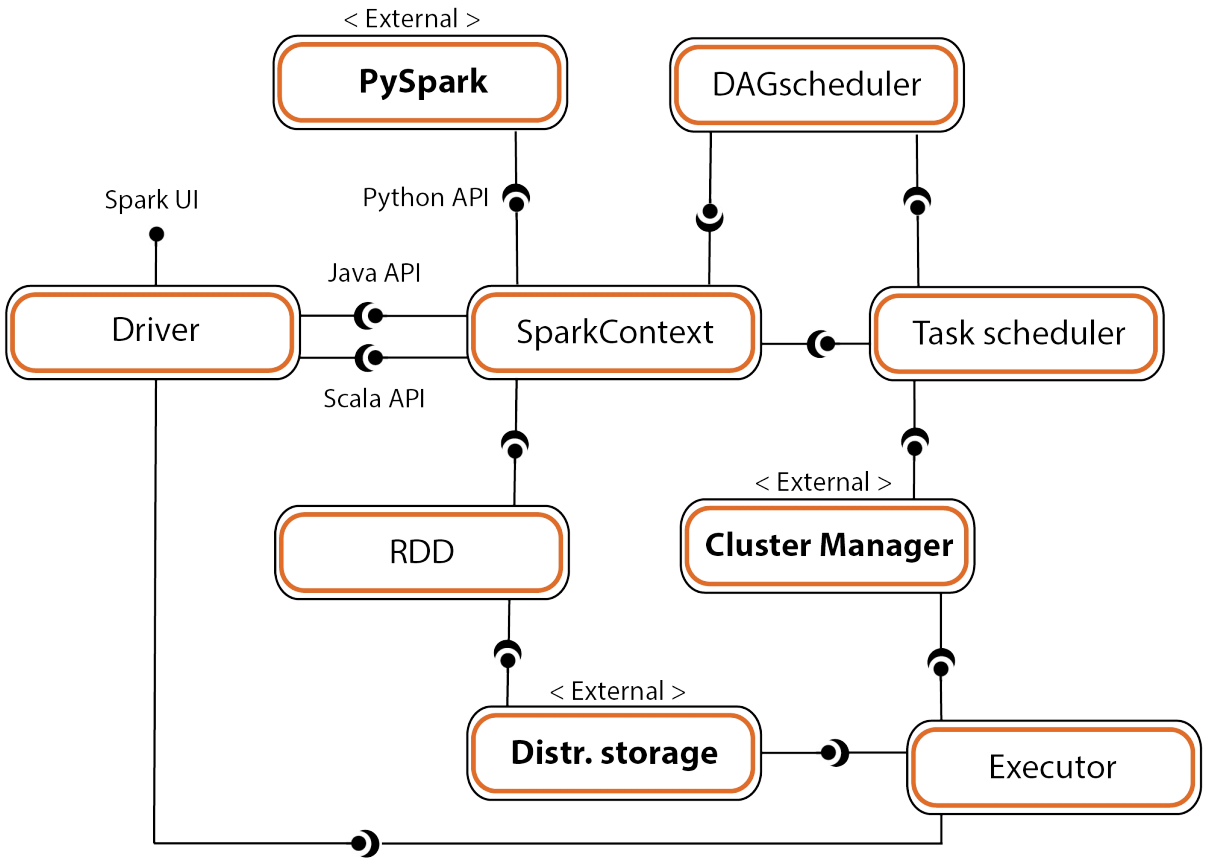
Apache HBase, *Amazon S3*, *Cassandra* and many others. These storage systems are the *data provider* to Spark and the expected data size can be up to petabyte level. Spark also supports a local mode, mostly for development and debugging purposes, where distributed storage is not required and the local file system can be used instead.
    - The **external interface** for these external entities is the *configuration of Spark*. Spark allows highly customized configurations so these file systems usually support Apache Spark by providing custom code that configures Spark correctly for their file system.
- **User**: Spark is used by various user applications from individual users to large-scale companies such as *eBay*, *Amazon*, *IBM*, *Netflix*, etc. These external entities make use of Spark through the Spark APIs, such as the RDD API which is for Scala. Spark also provides additional APIs for Java, R, and Python. These *APIs* provide the **external interface** to the users and can be considered *service consumers*. Like Spark itself, these service consumers are required to be scalable, fault-tolerant, and efficient.
- **Communication**: The main communication channel for the development of Spark is through *GitHub pull requests*, *JIRA* and *developers mailing lists*. Support for the users is provided through *StackOverflow* and a *user mailing list*. There is also a *chatroom* and there are frequent meetups.
- **Development Tools**: The project currently uses *GitHub* to develop and manage source code where pull requests are used to handle code reviews and merge code changes. *JIRA* is used as issue tracker where each issue is linked with Pull Requests on GitHub to ensure traceability.
- **Build Tool**: *Apache Maven* is used as a project management and comprehension tool in which the project is built and documentation is also handled. Support for *sbt* (Simple Build Tool) is also provided.

Functional View

A functional view describes the system's runtime functional elements and their responsibilities, interfaces, and primary interactions [5]. The functional requirements of Spark Core are identified as

> The main functionality for Spark Core is to provide an engine that is optimized for running parallel operations on distributed data. Additionally, Spark Core allows these operations to be defined in multiple programming languages and their status to be monitored.

The functional elements and interactions that are identified are illustrated in Figure 4.



When a user application starts, a *Spark driver* creates a *SparkContext*, with the Java API or Scala API as the interface. In the case where Python is used, *PySpark* is launched as the driver. *SparkContext* is the entry point of all the functionalities provided by Spark Core. It creates a *DAG Scheduler* and a *Task Scheduler* when it starts. *Resilient Distributed Datasets* (*RDDs*) are created by user's code on the data that the user wants to process. When an action is taken upon an *RDD*, *sparkContext* submits jobs to the *DAG Scheduler*.

An *RDD* contains data that is stored using *Distributed Storage* in a way that allows tasks to access it in parallel. Since *RDDs* implement lazy operations, an operation is executed only when an action is called. When this is the case, the *DAG Scheduler* finds a minimal schedule to run jobs, which is done by computing a directed acyclic graph (DAG) of parallel tasks called stages. It then submits the stages to the *Task Scheduler*. The *Task Scheduler* is responsible for submitting the tasks for execution to the *Cluster Manager*. The *Cluster Manager* then assigns *Executors* to run the tasks using the data in *Distributed Storage*. The *Executors* periodically send metrics of the running tasks back to the driver, which can then be accessed by the user through *SparkUI*. This way, the user can monitor his application.

To connect external elements, connectors are required. For the *Cluster Manager* and *Distributed Storage*, these correspond to the external interfaces identified in the Context view.

Performance and Scalability perspective

In this chapter, we provide insight into two related quality properties of Apache Spark: performance and scalability. These qualities are especially important as Apache Spark was created to provide better performance than Hadoop MapReduce [4].

Requirements

Given that Apache Spark is a framework for processing data, it behaves differently in response to heavy workloads than other systems. When there are fewer resources available, the response time and throughput will simply be lower for each processing task. The total response time and throughput for the system will still be as high as possible, which is acceptable for a data processing system like this.

In other systems where users might be waiting on individual tasks to complete, this might be unacceptable, as mentioned in the book by Rozanski and Woods [5]. This is an important aspect of Apache Spark as it heavily influences the requirements for performance and scalability. There are three aspects for which we quantify the requirements:

1. *Response Time*: The qualitative requirement is that the system should respond much faster than Hadoop MapReduce in use-cases like machine learning. Given the fundamental differences between Apache Spark and Hadoop MapReduce, we would quantify this as requiring a 10x improvement in response time over Hadoop MapReduce, given the same processing work and a sufficient amount of available worker nodes. This should indicate that Spark is intended to provide a significant performance improvement over Hadoop MapReduce.
2. *Throughput*: The throughput is difficult to specify since the type of work that can be done in Spark is highly variable as it is determined by the code provided by the user. Of course, in order to achieve the required improvements on the response time, the throughput will need to be significantly improved.
3. *Scalability*: While not explicitly mentioned, it's safe to assume that the scalability requirement for Apache Spark is to at least scale as well as Hadoop MapReduce.

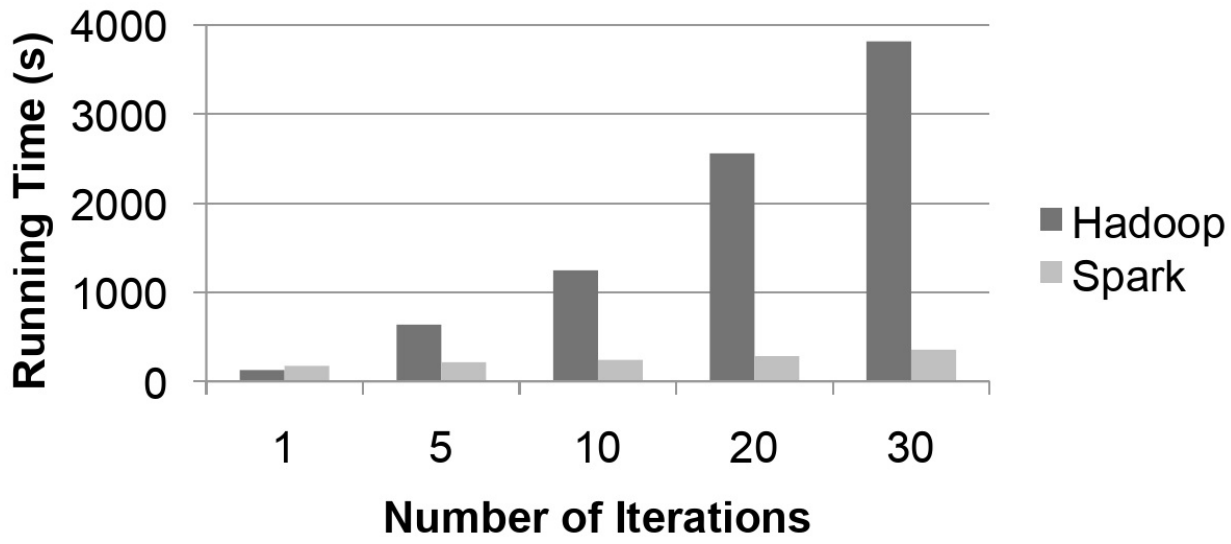
Performance

The performance of the system is typically determined by the response time and throughput characteristics. We only have a qualitative requirement on response time. In order to address this, Apache Spark was designed to use RDDs. An RDD is a collection of objects which are partitioned across machines. The next part shows how it improves both the throughput and response time of a single task being executed by Spark.

An RDD allows the data it contains to be accessed multiple times. This allows a significant improvement over Hadoop MapReduce which only allowed the data to be accessed once for every time it is loaded from disk. This improved the throughput of the system as a task can simply revisit data as needed, allowing more work to be done within a given time period.

By allowing the RDD to be cached in memory, the response time is significantly reduced as the data is available much more quickly. This characteristic combines especially well with the improvement to the throughput. This causes the total response time to be improved even more.

When Spark was created, this performance was benchmarked using a logistic regression that was run with different amounts of iterations (Figure 5). It shows that the response time for the initial implementation of Apache Spark had already achieved a 10x improvement, which is especially noticeable with the higher amounts of iterations. In the latest version of Apache Spark, this improvement was up to 100x, according to their website [1].



In addition, RDDs achieve fault tolerance through a notion of lineage by having the ability to rebuild RDDs even if a node is lost during operation. This means that even when nodes fail entirely, the task can be completed. This was measured as causing only a 21% decrease in response time [1].

In general, we have noticed that several of the tactics proposed by Rozanski and Woods [5] have been applied in Apache Spark. Most notably, they have minimized the use of shared resourced by only allowing non-exclusive access to the data. Being a distributed system, they allow scaling out to improve performance. And since it is a distributed system, they also partition and parallelize the work by splitting them into tasks which can be run on separate nodes.

Scalability

Scalability is the ability of a system to handle an increased workload. It determines whether Spark Core is still functional and fast when it has to process larger amounts of data. Nowadays, Spark has been shown to work well with up to petabytes of data [12]. It does this by allowing more nodes to be added to the cluster to process this data. This has been shown to work well with up to 8000 nodes [12]. This indicates that the amount of efficiently participating nodes is an important metric for scalability.

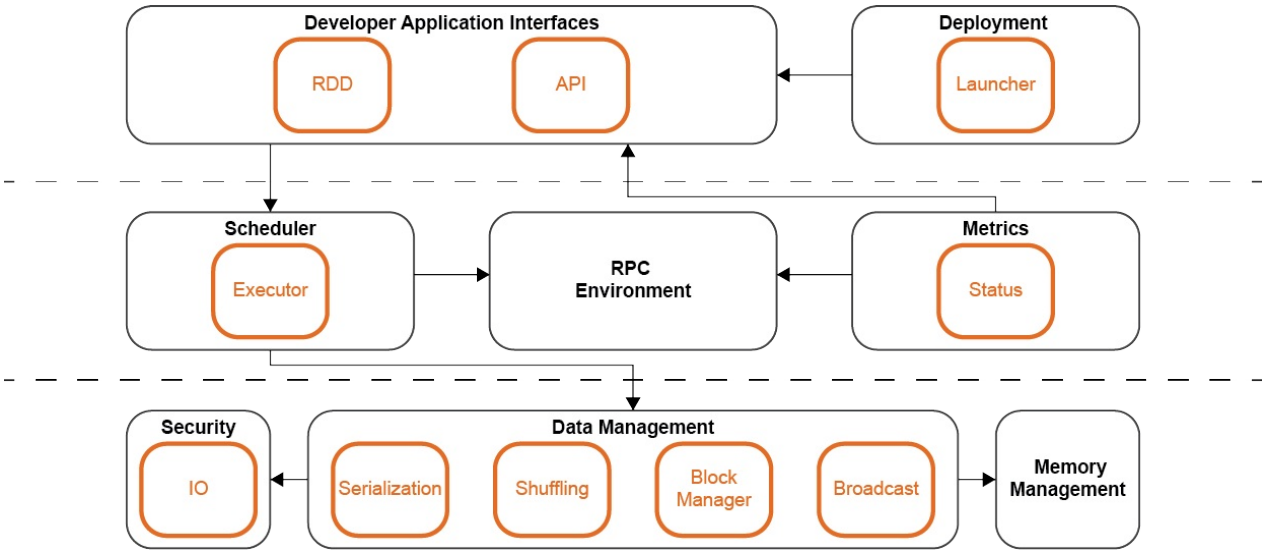
As mentioned earlier, the scalability requirement for Spark Core is to at least scale as well as Hadoop MapReduce. In order to achieve this, Apache Spark was created to use the same components as Hadoop for its cluster management and distributed storage. This means nodes are managed in the same way as with Hadoop MapReduce, just as the data storage. With both the nodes and data being managed by the same components, this should provide Apache Spark with a similar level of scalability as Hadoop MapReduce.

Development View

This section discusses the development view of the Spark Core. It provides an overview of the architecture that supports the software development process. This is largely based on the code of Spark Core in GitHub but also on the documentation [10] and website of Spark [1].

Module Structure Model

To facilitate the understanding of Spark Core, we provide a structural model that depicts the overall organization and distribution of modules within the system. While we were able to identify some layers based on the interaction between modules and the abstraction level of the modules, they do not seem to be created by design. This is why they don't have any name, though you may consider them as the *Top*, *Middle*, and *Bottom* layer. There do not seem to be any layering rules designed for these layers. This is not strange since the layers weren't created by design but emerged gradually.



Spark Core is organized as a set of modules, each providing specific functionalities. The diagram in Figure 6 shows these modules, their submodules and the dependencies between them. Here we describe each of the modules in a bit more detail:

- **Developer Application Interfaces**: This module provides the access to the functionalities available to users in Spark Core, most of which is contained in this module. In addition, the API submodule provides programming interfaces for other programming languages. This allows Spark Core to be used natively in multiple programming languages. It interacts with the Scheduler to schedule the tasks that need to be executed.
- **Deployment**: This module handles the Spark deployment, also known as *run modes*. It provides the required functionalities to

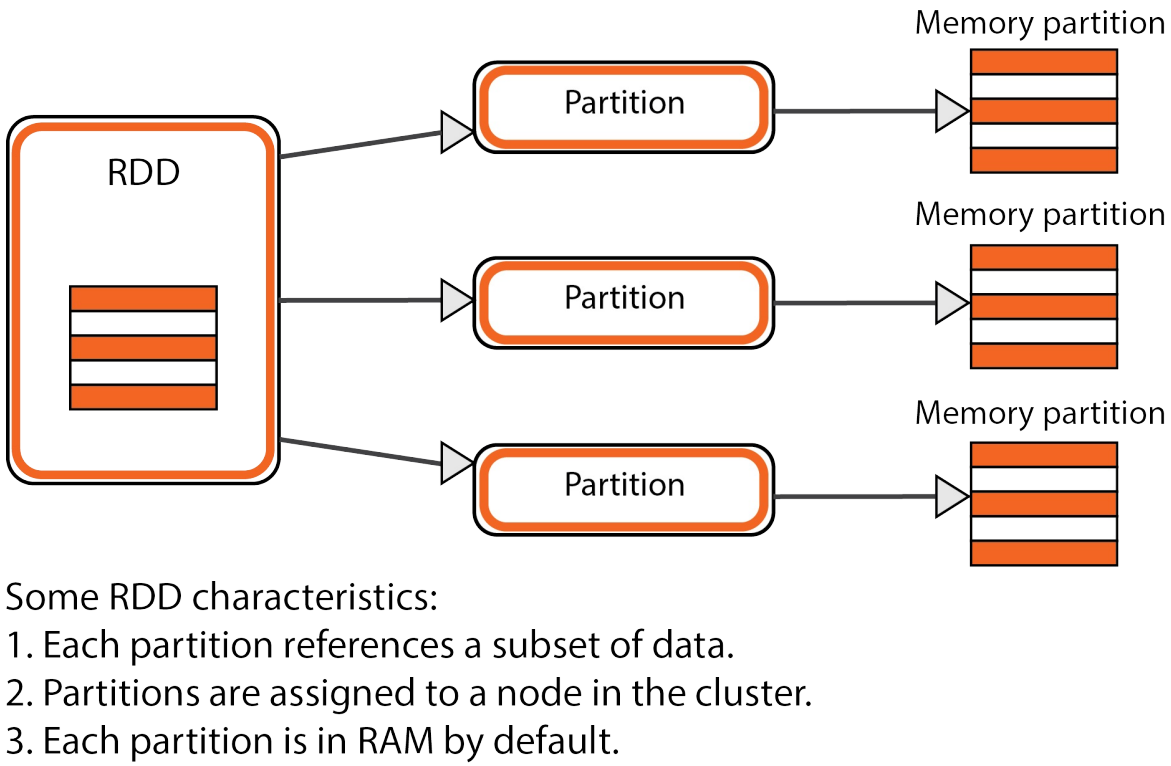
support a local or clustered application. In addition, the launcher submodule is provided as an interface to launch Spark applications programmatically. It interacts with the Developer Application Interfaces to access the Spark functionalities.

- **Scheduler:** This module takes care of delivering and scheduling tasks among workers in a cluster. It interacts with the RPC environment to send messages to where they need to go.
- **Metrics:** Spark uses the Metrics 3.1.0 Java library to provide insight into Spark instances such as the Scheduler and Executor. These metrics are only available when using Spark as a clustered application. It is automatically turned off in applications that only run locally.
- **RPC Environment**: This module facilitates an environment to process messages. It provides functionalities such as routing incoming messages, stopping messages and registering endpoints. It interacts with Data Management to process the messages.
- **Data Management:** This module handles most of the lower-level functionalities related to data. These are the functionalities that are required to make RDDs work. In addition to this, this module also includes serialization functionalities to increase efficiency on data transmission. It interacts with Security to ensure that the data is secured correctly. It also interacts with Memory Management to handle any data that needs to be in memory, like caching RDDs.
- **Security:** The security module provides IO encryption and decryption streams to secure the data.
- **Memory Management:** This module handles the functionalities that Spark requires when working with memory.

Common Processing Model

Isolating common processing across elements into separate code units contributes to the overall coherence of the system and reduces duplication. We identified 2 elements that fit this description.

- **Common data structures:** Apache Spark is built around a single common data structure, the RDD [4]. As the name implies, it is a read-only dataset that is distributed across multiple machines and can be rebuilt if part of it is lost. This data structure is what provides the increased performance of Apache Spark when compared to a MapReduce engine like Apache Hadoop in certain cases. As such, it is clear that this is a common design element that is fundamental to the architecture of Apache Spark, which has been designed around it. The entire Spark Core module is created in order to provide the common software that allows RDDs to be used consistently.



- **Use of third-party libraries**: Apache Spark is designed to use third-party libraries whenever possible, which we consider a standard design approach. Specifically, we can see that the cluster manager and distributed storage system have been created in a way that allows them to reuse components from the Apache Hadoop project, namely YARN and HDFS. But the cluster has been designed in a way that allows other third-party libraries to be used as well, like Apache Mesos instead of YARN or Hive instead of

HDFS. So some common processing is provided to abstract away the actual libraries.

Technical Debt

Technical debt is what occurs when developers have to make a compromise between making the code work perfectly and making it work at all. The latter can usually be done with fewer resources and in less time but may cost more time down the road. This is the debt that can incur when the resources or time that is needed cannot be spent right away.

In this chapter, we analyze and discuss how debt is handled by the community.

Handling Technical Debt

We've looked into how the Apache Spark developers deal with technical debt. For the most part, they try to keep technical debt as low as possible when they accept new code. They do this by using a code analysis tool to check the code from a Pull Request before they merge it. Specifically, they use Scalastyle as code analysis tool on Jenkins which they've integrated with Github. This allows them to verify several things automatically and ensures that any merged code is of sufficient quality. Of course, this doesn't remove technical debt altogether. For one thing, the developers sometimes disable certain rules because it's simply too costly to fix all the identified problems. However, they keep this in mind and try to work towards enabling these rules again. There is no clear process for this though, it's just something they keep in mind.

They also handle technical debt through their normal development procedure. If they identify technical debt that needs to be fixed, they will create an issue in JIRA to register this. After this, it can be fixed if someone has the time and expertise to do so. This means that it can remain open for quite a while like in SPARK-2296. However, this isn't always the case. Looking at SPARK-3453, we see that the issue was created in September 2014 and the corresponding Pull Request was actually already merged in October 2014. This shows that technical debt can be solved quite quickly too.

The developers do think of quality, so it is common to observe discussions regarding how to correctly implement things in Pull Requests and issues. But all discussions related to technical debt seem to be done through the mailing list. As an example, we found a discussion on the mailing list about the future of Python 2 support within Spark. This discussion was motivated by the fact that Python 2's end-of-life has been scheduled so Python 3 must be supported before then. The support of both Python 2 and 3 could introduce a large amount of technical debt.

One area where we have noticed that they are still lacking though is that they don't seem to actively try to identify areas with technical debt. Most of the technical debt we've seen they discuss has been found incidentally or has been triggered by outside influences like platforms or libraries. This means that they may have technical debt they are unaware of.

Conclusion

Apache Spark was created to achieve high performance on large-scale data, and this has been reflected in the whole architecture of Spark Core, which builds around RDD. Throughout our analysis, we found that Spark Core has well-designed interfaces that allow APIs and functionalities to be added. The fact that it reuses existing libraries and projects also allows it to achieve good scalability right away. Moreover, we found that developers of Spark are very careful about introducing technical debt. This results in a clean codebase. Due to the open-source nature of the project, developers from various companies, carrying out various use cases, contribute directly to the project. These factors make the development of new features in Spark fast.

Throughout each release of Spark, continuous efforts in supporting modern cluster managers and complex data sources, as well as further optimizing performance and stability, are observed. We believe that the architecture and the way of working in Apache Spark would allow it to meet the needs of future big data applications.

Reference

[1] Apache Spark Homepage. https://spark.apache.org/

[2] The Apache Software Foundation Blog. https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces50

[3] The Apache Software Foundation Project Statistics. https://projects.apache.org/statistics.html

[4] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets

[5] Nick Rozanski and Eoin Woods. 2012. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, 2nd edition.

[6] Apache Spark Community. https://spark.apache.org/community.html

[7] Gigaom on Spark and Hadoop. https://gigaom.com/2014/06/28/4-reasons-why-spark-could-jolt-hadoop-into-hyperdrive/

[8] Apache Software Foundation page about Project Independence. http://community.apache.org/projectIndependence

[9] Matei Zaharia. 2014. An Architecture for Fast and General Data Processing on Large Clusters (Ph.D. Thesis)

[10] Apache Spark Documentation. https://spark.apache.org/docs/latest/

[11] The Apache Software Foundation. http://apache.org/

[12] Reynold Xin, https://www.slideshare.net/databricks/large-scalesparktalk.

[13] Apache Spark @Scale: A 60 TB+ production use case https://code.facebook.com/posts/1671373793181703/apache-spark-scale-a-60-tb-production-use-case/.

[14] Renan Souza, Vítor Silva Sousa, Pedro Miranda, Alexandre A. B. Lima, Patrick Valduriez, Marta Mattoso. 2017. Spark Scalability Analysis in a Scientific Workflow

TypeScript: typed JavaScript

By Taico Aerts, Chiel Bruin, Maarten Sijm and Robin van der Wal
Delft University of Technology

Official TypeScript Logo (on GitHub [1])

Abstract

TypeScript is a programming language that brings types to JavaScript. TypeScript, in contrast to JavaScript, gives developers a lot of feedback on their code and detects bugs well before the code is deployed. It is a language that focusses on usability, which is reflected in the language design, the IDE integration but especially in its popularity. Currently ranked as the third most loved programming language, TypeScript is well on its way to overtake JavaScript as the web language of choice.

In this chapter, we present a detailed analysis of the architecture of TypeScript from various viewpoints. Furthermore, we provide a perspective on the usability of the language. Finally, we close with an analysis of technical debt. We conclude that TypeScript has a well-designed architecture and has very little technical debt.

Table of Contents

Introduction

In this chapter, the architecture of the software system TypeScript is examined and evaluated. But before we start, we will state the purpose, capabilities and history of TypeScript. To quote their website [2]:

> TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. Any browser. Any Host. Any OS. Open Source.

The main functionality is, in other words, to provide an extension of the JavaScript language to support types. While typing can make a language feel more restrictive, it also has many advantages, for example in terms of maintainability and readability of the code. Typing also enables more powerful static analyses, which allows for the detection of mistakes at compile time instead of at runtime and decreases debug time [3].

TypeScript compiles to regular JavaScript, which means that it works on most browsers and operating systems. It offers support for the latest ECMAScript standards and is supported by multiple IDEs. An added bonus is that it is open source and free to use. TypeScript is thus suitable for developing large scale web applications and is used in many existing applications [4].

TypeScript started as an internal Microsoft project in 2010. After two years, the basic functionality was completed and the project was made open source [5]. TypeScript was praised for its typing functionality back then but missed support from any IDE. This was followed up by adding IDE support and editor support in 2013 [6]. In 2016, TypeScript version 2.0 added more type functionality, such as more control over null types [7]. Currently, TypeScript is still in active development on its GitHub repository, where hundreds of contributors have contributed to the project [8].

Section 1 analyses all stakeholders of TypeScript to determine which people are most important for its development. Next, Section 2 provides a context view of TypeScript. In Section 3, a functional view of TypeScript is given which is complemented by a development view in Section 4. Section 5 follows up with an analysis of TypeScript from a usability perspective and technical debt is examined in Section 6. Finally, Section 7 gives an overall conclusion on the architecture of TypeScript.

1. Stakeholder Analysis

In order to identify the relevant stakeholders, we started by analyzing different sources related to the system. In Section 1.1, we investigate the use of issues and pull requests on GitHub and in Section 1.2, we investigate external websites. This gave us a clear overview of the people who have interest in the system. This overview is shown in Section 1.3, followed by an overview of the power and interest of these stakeholders in Section 1.4.

1.1 Issue and Pull Request Analysis

The first source we analysed is the official GitHub repository [8]. From issues and pull requests on this repository we were able to identify the key developers, their workflow and other stakeholders with interest in the system.

We found that the main development is done by a core group of Microsoft employees. They make many changes to the code and review the contributions made to the project via pull requests. In addition to this, they maintain the roadmap of the project in two ways: they manage the issues and milestones of the project, and they publish notes of team meetings as issues on GitHub.
A last role for the development team is helping users that have opened an issue with problems they encountered. Sometimes a pointer to the documentation helps to resolve the problem, but often these issues unveil an actual problem with (a newer version of) the implementation.

We were also able to identify other stakeholders like the continuous integration providers that build TypeScript, JavaScript frameworks that use TypeScript in their development and IDEs that integrate with the language.

1.2 Analysis of External Websites

Besides the pull requests and issues on the official GitHub repository [8], we also investigated external sources in order to identify the stakeholders. Guided by the sources that are mentioned on the TypeScript website [2], we looked at StackOverflow [9], Twitter [10], the Microsoft Developer Network (MSDN) Channel 9 [11], the TypeScript blog [12] and a list of languages that compile to JavaScript on the CoffeeScript GitHub [13].

From these sources, we found that part of the core development team actively engages with the community by answering questions on StackOverflow and by writing blog posts detailing new features of each release. These releases are announced via both the official TypeScript Twitter account and the GitHub repository. Lastly, we identified other people on StackOverflow that actively helped people with their problems, despite not being affiliated with Microsoft. Some of these people have even written books on how to use TypeScript.

1.3 Stakeholders of TypeScript

Rozanski and Woods [14] identify ten types of stakeholders. In our classification of the identified stakeholders, we mostly followed this classification, only making some small changes to better fit the project. The stakeholder types that are the most relevant to TypeScript are shown in Table 1.1.

Type	Relevant actors
Acquirers	Oversee the procurement of the system. Hold the final rights to the project
Developers	Construct, test and deploy the system from the specifications
Maintainers	Manage the evolution of the system over time
Assessors	Oversee the system conformance to standards (code, documentation and specifications) and legal regulation. Can be seen as integrators in this project
Communicators	Explain the system to other stakeholders via its documentation and training materials, also keeps other stakeholders up to date to system changes and releases
Suppliers	Build and/or supply hardware, software, or infrastructure on which the system will run and deploy
Support staff	Provide users support for the system when it is running
Users	Define the system's functionality and ultimately make use of it
Competitors	Aim to put a similar, competing system on the market

Table 1.1 - *Relevant stakeholder types for TypeScript and their roles*.

Many stakeholders have multiple roles in the project, making them part of multiple of the different types. Therefore we created an Euler diagram that shows all the relevant stakeholder types and the actors they contain (Figure 1.1).

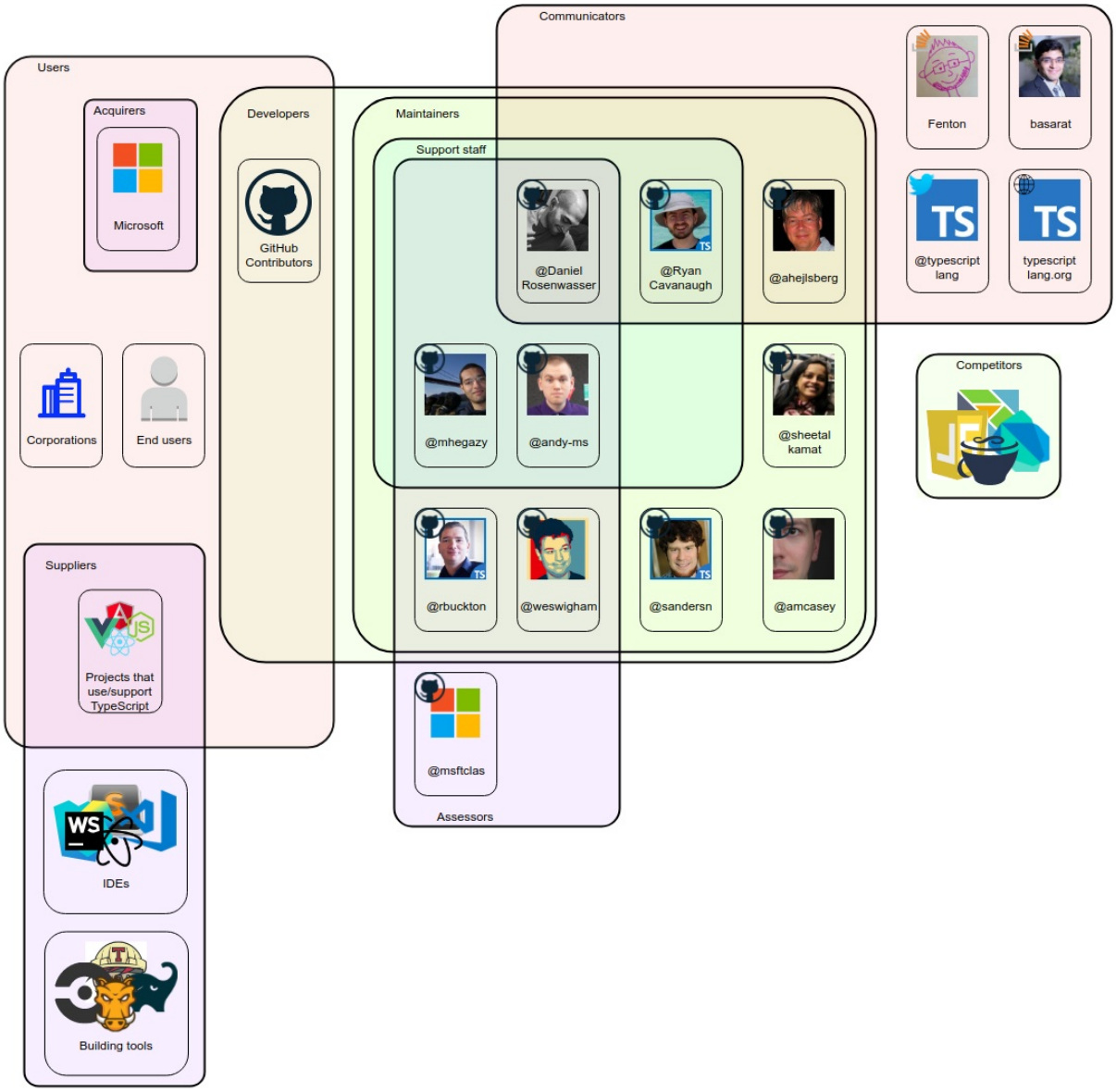


Figure 1.1 - *An overview of the stakeholders of TypeScript and their roles.*

1.4 Power-Interest Grid

Another way in which we classified the stakeholders is by looking at their power in changing the system and the interest they have in the system. Figure 1.2 shows this relation in a grid with the relative power and interest the different stakeholders have in TypeScript.

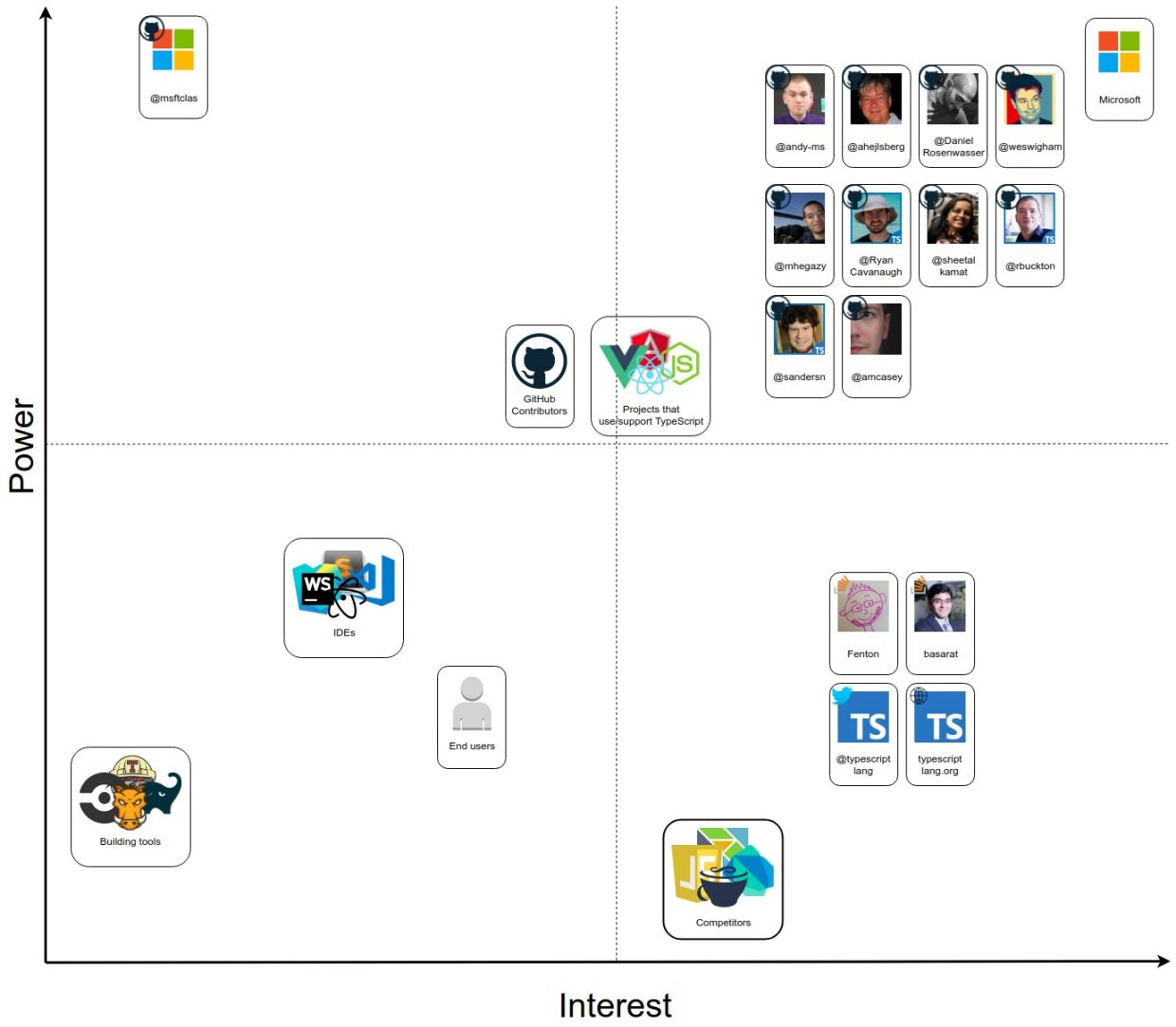


Figure 1.2 - *The power-interest grid showing the different stakeholders.*

1.4.1 Microsoft Development Team

As the owner of the project, Microsoft has the most power and interest in the system. The development team employed by Microsoft to work on TypeScript have slightly less power and interest than their employer, but they guide the main development of the system.

1.4.2 Communicators

The communicators are a group of stakeholders that have high interest in the system, but who lack the capability to steer the direction of the project. This group includes the writers of books on TypeScript, the StackOverflow users and the official announcement channels. In order to fulfil their roles as communicators, they need to be up to date with the state of the system. This gives them a great interest in the system but does not give them any power over it.

1.4.3 Competitors

Competitors have a reasonably high interest in the system. This is because they want to have a better system themselves to gain a competitive advantage over TypeScript. To achieve this, they need to closely watch the development of any new features that might give TypeScript this advantage. Following this, competitors have a small amount of power in the development of TypeScript with the features they introduce.

1.4.4 Users

Users generally have little power over the project, but since TypeScript is an open source project, users can submit issues and make contributions. As such, we have divided users into two different groups: those who make contributions and those who do not. Generally, a contributing user has a larger interest in TypeScript than a user that does not.

1.4.5 Tooling

The last two groups in the power-interest grid are the build tools and IDEs. These have relatively low power and interest in the system as they do not directly depend on TypeScript. The IDEs are a little more dependent on it as their integration with the language can be a reason to use this IDE over another. They also have more power as many users will use TypeScript via one of these IDEs and are therefore an important factor in the adoption of the language.

2. Context View

The context model in Figure 2.1 provides an overview of the external entities related to TypeScript. The following sections will explain each of the external entities in clockwise order, starting from the left.

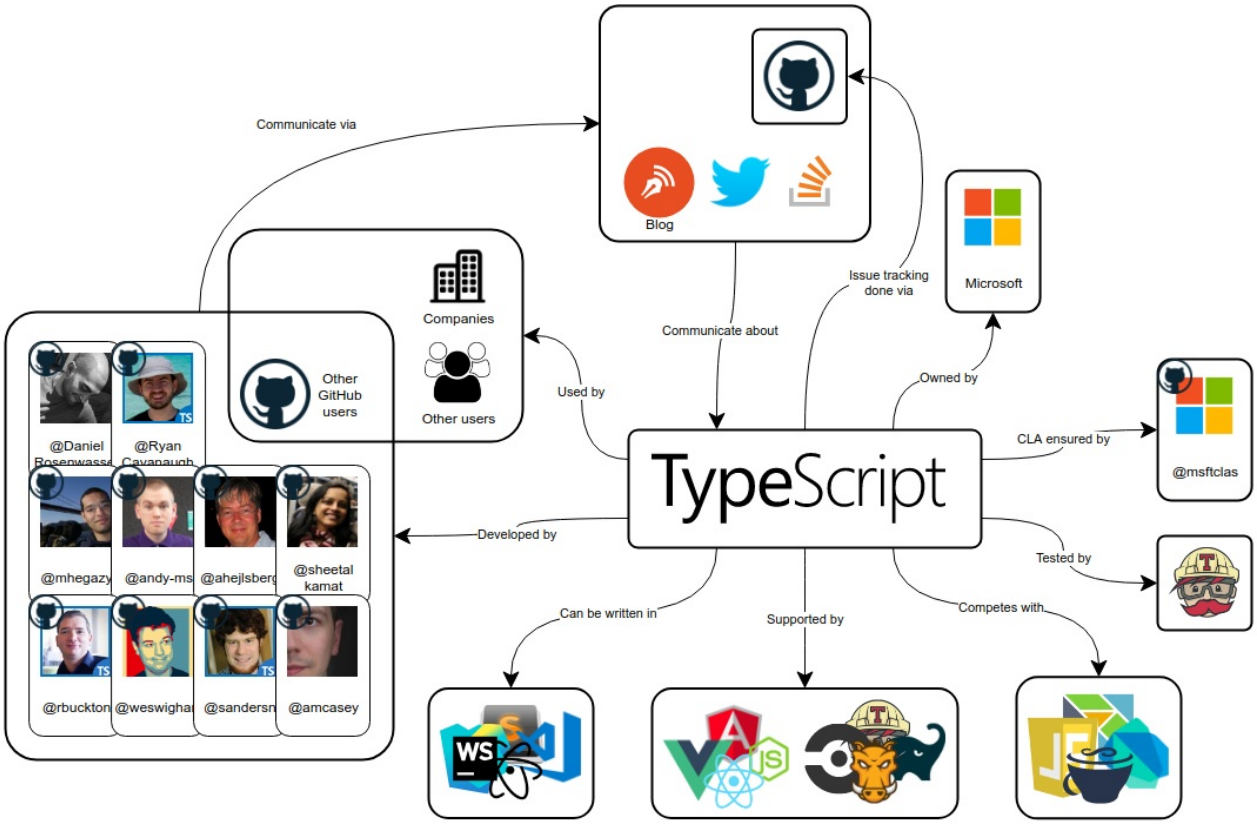


Figure 2.1: An overview of the external entities related to TypeScript.

2.1 People: Developers and Users

On the left in Figure 2.1, the active developers of TypeScript are shown. They are employed by Microsoft, the owner of TypeScript. They are not the only developers, however: there are GitHub users from all over the world that contribute to the project, mostly by submitting pull requests to the repository. These GitHub users also form a part of the group of entities that use TypeScript, among companies and general other users.

2.2 Communication Channels

The top of the context model shows the four primary communication channels for TypeScript.

- The most important communication channel is GitHub. This is where the issue tracking is done, pull requests are reviewed, meetings are documented, etcetera.

- The TypeScript blog and Twitter are used to broadcast news about TypeScript such as newly published releases and other important events.
- StackOverflow is used by users to ask questions about TypeScript, which are often answered by members of the development team.

2.3 Quality Ensurance

On the right in Figure 2.1, two tools are shown that TypeScript uses to assess the submitted pull requests.

- TypeScript uses a Continuous Integration bot that checks whether the user that submitted the pull request has signed the CLA of Microsoft.
- Travis CI is used to execute all tests in the repository, to make sure that the submitted pull request does not break any existing functionality.

2.4 Competitors

At the bottom right of the Context Model, the competitors of TypeScript are shown, that were discussed in Section 1.4.3. These competitors include the Elm language, Dart, CoffeeScript, and JavaScript.

2.5 Suppliers

The remaining external entities correspond to the Supplier stakeholder type, as was discussed in Section 1.3. The following entities are shown in Figure 2.1:

- Build tools: Travis CI, Gradle, Grunt, and CircleCI
- Frameworks: Angular, NodeJS, React, Vue.js
- IDEs: Sublime Text, Visual Studio Code, Atom, WebStorm

3. Functional View

Every software system has some functionalities that are exposed to the user. These functionalities can be modelled as part of the architecture of TypeScript. To this end, this section shows a Functional View as described in Rozanski and Woods, Chapter 17 [14]. Section 3.1 describes the capabilities of TypeScript, while Section 3.2 shows the external interfaces of TypeScript that are exposed to the user.

3.1 Capabilities

TypeScript has two main capabilities.

1. **Compile** TypeScript files to plain JavaScript files [2].
   This capability of TypeScript is fully described in the language specification on GitHub [15]. This language specification is an exhaustive document that describes all possible constructs in the TypeScript language.
2. Provide **editor-like functionalities** [16].
   This consists of many small capabilities, including code completions, code formatting, refactoring, debugging, incremental compilation, etcetera. It is also possible to write custom language service plugins [17].

3.2 External Interfaces

Both capabilities defined in Section 3.1 have their own external interfaces. In the following two sub-sections, we describe these interfaces and their responsibilities.

3.2.1 TypeScript Compiler

The first capability is provided by a standalone compiler. It compiles TypeScript source files to JavaScript files, in such a way that the emitted JavaScript files resemble the TypeScript input, as described in the introduction of the language specification [15].

The compiler is a command-line tool that has many options [18]. All of these options can also be provided in `tsconfig.json` files in any directory that contains TypeScript files. Using a configuration file makes it easy to use compiler options consistently in a project. With a valid configuration present, the TypeScript compiler can be invoked without arguments and it will use the `tsconfig.json` file in the current directory [19].

3.2.2 TypeScript Server: Editor Functionality

The editor-like functionalities are provided by a standalone server. This server is a wrapper for many capabilities related to editor functionality, as listed in Section 3.1.

The TypeScript server communicates using JSON messages [20]. However, typical users of TypeScript will not directly interface with the server. Most IDEs include the standalone server and use it to provide static code checks, possible refactorings and other things in a user-friendly way.

4. Development View

In this chapter, we will present a part of the Development View as described in Rozanski and Woods, chapter 20 [14]. Section 4.1 provides a module structure model of TypeScript. Section 4.2 continues by describing to what extent the commonality of design between modules, code and organisation is achieved throughout the project. We have also analysed the codeline models of TypeScript, but those have been left out as they do not provide important additional information on the architecture of TypeScript.

4.1 Module Structure

In this section, we provide a module structure model of TypeScript. A module structure model shows how the system is organized into modules and what the dependencies between these modules are.

We have identified the main TypeScript modules and organized them into four layers, as shown in Figure 4.1. These layers are:

- **Compiler layer**: The compiler layer is responsible for compiling TypeScript programs into JavaScript.
- **Language Service layer**: The language service module wraps the compiler layer and provides editor-like services like formatting, refactoring, code completion and debugging.
- **Server layer**: The server exposes the language services and compiler features to users through a JSON protocol. Editors and IDEs can interface with the server to use the language services and to compile code.
- **Test Harness layer**: The test harness contains various test runners to execute different types of tests. It also contains test transformers which change the code in test files to help the test runners.

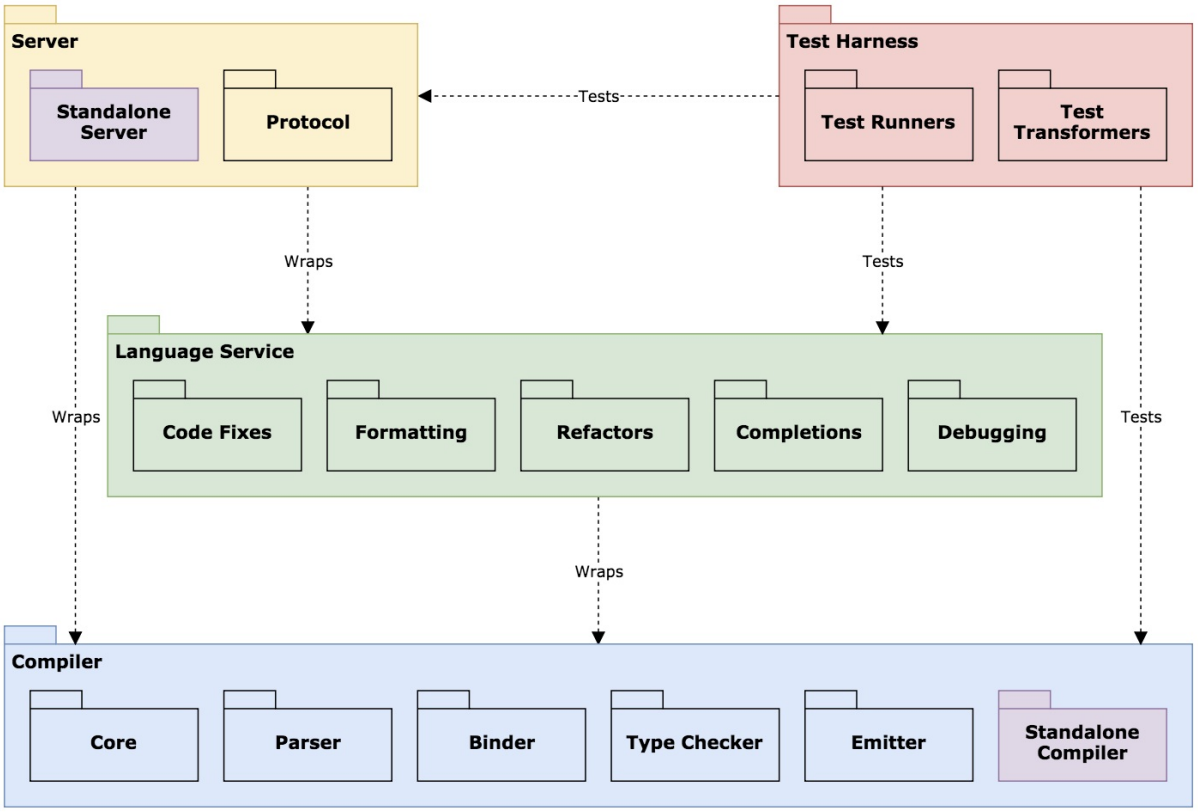


Figure 4.1 - *Module Structure Model of TypeScript, showing the different layers and the dependencies between them.*

In this overview, each module represents one or multiple source files which together perform a similar role in the system. Please note that we have grouped together some modules to prevent the image from becoming too detailed. The purple-coloured modules are modules that a user of TypeScript can interact with. All modules that are not purple are internal to TypeScript.

We can see from the module structure that the different capabilities, as defined above in the Functional View (Section 3), reside in separate layers in the repository. The editor functionality is even split into two layers: one layer that contains the editor services and one separate layer for the server, which wraps the compiler and language service layers.

4.1.1 As-designed versus As-implemented

TypeScript provides a layer overview [16] on their wiki. We have included this layer overview as Figure 4.2. While this image isn't exactly the same as a module structure model, it does describe similar layers to our module structure model.

One big difference is that the test harness is not shown in the architectural overview. While the test harness did exist whenever the image was created, it was probably left out since it is not involved in the compilation/static analysis pipeline.

Another difference is that "VS Shim" is mentioned on the layer with the server. In the code, the "shims" are in the Language Service directory and seem to offer functionality similar to other language services. For this reason, we decided to leave it out to keep our module structure simple.

Besides the layer overview, TypeScript also provides an overview of the common data structures that are used to communicate between layers. The most important one of them is the Abstract Syntax Tree (AST). Looking at the code in the different layers, it looks like the interface for an AST node is defined in multiple ways, in this way exposing different functionalities of nodes to different layers.

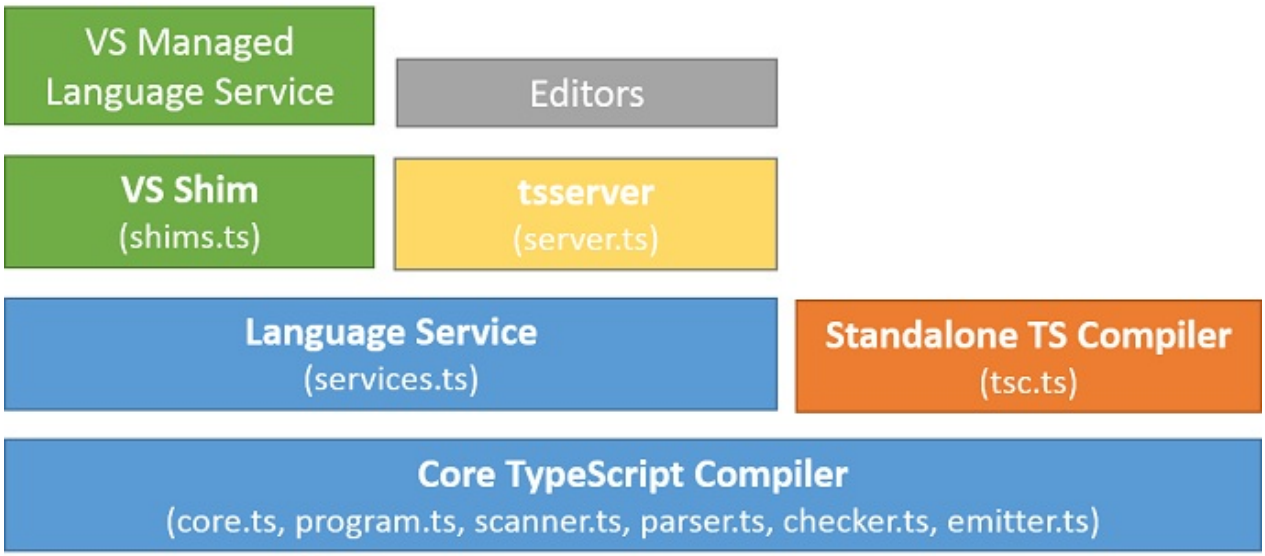


Figure 4.2 - *Layer overview from the TypeScript wiki. [16]*

4.2 Common Design

In our analysis of TypeScript, we have found some common elements in the design of TypeScript. For the brevity of the chapter, we have left out the complete analysis in our chapter. The most important findings of common design elements are:

- Internationalization, logging, debugging and performance metrics are standardized across the project.
- TypeScript has a clear standardized design methodology and testing procedure when it comes to contributions and changes.
- Standard code solutions like design patterns are used sparingly. It seems that TypeScript avoids common design patterns because of a focus on performance. This is detrimental to the readability of the code as different pieces of code performing a similar task can look very different. It can also make it more difficult for someone to contribute to the project because the developers might want to see particular code patterns, but these patterns are not communicated to contributors.

We can see that TypeScript uses common design of elements wherever this applicable, in order to prevent code duplication as much as possible.

5. Usability Perspective

In this section, we analyze TypeScript from a usability perspective. For TypeScript, usability means something completely different than for other software systems. Its usability is not decided by the user interface of the software. Rather, the usability of TypeScript is determined by the usability of the language and by the ecosystem around TypeScript. This concerns architectural elements like the expressiveness of the language and the understandability of compiler output, but also elements like tutorials, support with questions and integration with IDEs. To analyse this, we first look at different usability aspects in TypeScript itself. Then, we take a look at the ecosystem around TypeScript.

5.1 Usability Aspects in TypeScript

5.1.1 Internationalization

Internationalization is used for the different error messages that can be generated by the compiler. To change the language, users have to set the `--locale` flag to the language of their choice, e.g. `--locale de` for German. The locale can also be configured globally with a `tsconfig.json` file. TypeScript currently supports 14 different languages, covering the most used languages on the planet. Usually, it is difficult for a project to keep the translations up to date when messages are added or changed. However, TypeScript is localized by the Microsoft localization team which updates the localizations periodically. For TypeScript this means that localizations are almost always up to date. As TypeScript also supports UTF-8 and UTF-16 in its parser and compiler, users can easily use TypeScript in the language of their choice.

5.1.2 Conformance Testing

TypeScript is a superset of JavaScript. As such, they want to support all JavaScript functionality. New JavaScript functionality is added from time to time in the form of new ECMAScript standards. TypeScript also wants to conform to these standards, both in the original TypeScript code as in the JavaScript code it compiles to. To ensure this conformance, TypeScript has multiple tests for the ECMAScript functionality. Any change that would break this conformance is detected and will be modified. By supporting all this functionality, users can continue to use the features from JavaScript that they are familiar with and will probably have an easier time to switch to TypeScript.

5.1.3 IDE Integration

TypeScript is supported by multiple IDEs and editors, including Visual Studio, Eclipse and WebStorm, but also Sublime, Atom, Emacs and Vim. Users thus have a wide variety of choice in which editor they use. TypeScript provides the IDE functionality with the TypeScript server, which is then used by the IDE or editor. For users, this means that the experience is consistent across the different supported editors.

5.2 The TypeScript Ecosystem

5.2.1 Documentation and Support

TypeScript offers extensive documentation on its website. There are tutorials for people who are new to TypeScript, giving quick and simple explanations on how to get started. There is also a handbook which contains much more detailed explanations of the different features. A formal language specification can be found on GitHub. For further questions, users are referred to StackOverflow, where questions are answered by an active community and sometimes also by the developers of TypeScript.

We do see that the handbook is not up to date with the latest functionality. There is a section "What's New" that discusses new language features in detail, but these features should also be added into the handbook, e.g. with a note in which version they have been added.

5.2.2 User Satisfaction

The best way to determine usability is by asking the users themselves. Every year, StackOverflow has a survey amongst its users, with questions regarding popular technologies and languages [21]. With responses from over 64.000 developers from all over the world, it is the largest survey in the field. In last year's survey, TypeScript was the 9th most popular language across all categories and the 7th most popular language according to web developers. In addition, it ranked 3rd on the list of most loved programming languages. These results suggest that users are pretty satisfied with TypeScript.

6. Technical Debt View

Software developers can sometimes decide to implement something the "quick and dirty" way. Such a solution can be justified for many reasons like time pressure but is often not the best solution when looking at code quality. In such a case, the developers deliberately introduce so-called technical debt. This debt can accumulate over time and can be decremental to the maintainability of a system. Therefore, developers need to pay attention to their technical debt every once in a while, in order to raise the quality of the code again.

In this section, we will investigate the technical debt that is present in the code of TypeScript. We will start with a static tool analysis of the repository, then we analyze the "todo" comments left in the code base followed by analysis on the testing debt. Finally, we look at the evolution of technical debt.

6.1 Static Tool Analysis

TypeScript is a project of 135k lines of code. This is too much to go over manually. Instead, a static tool analysis was done on the project using SonarQube [22] with the SonarTS plug-in [23]. The overview of this analysis can be seen in Figure 6.1. Note that for testing we run a different tool, this can be found in Section 6.3.

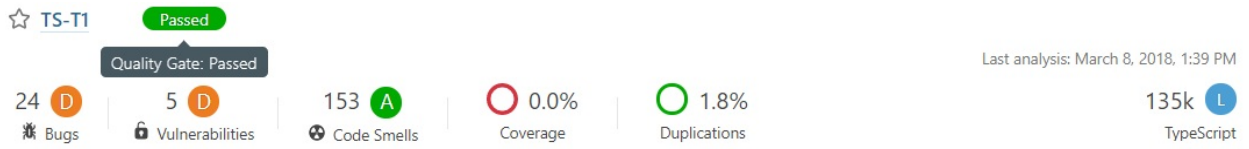


Figure 6.1 - *Overview of SonarQube after running the tool on the TypeScript project.*

According to SonarQube, TypeScript has a low score for bugs and vulnerabilities. However, most bugs and vulnerabilities are false positives since TypeScript is a compiler, which requires some workarounds in the code. The most common bug is useless self-assignment, which could easily be fixed.

Next, the code duplication is small and the code smells are not severe. Most duplications take place in non-critical parts of the repository, such as the testing harness or the server hosting classes, there is no code duplication in the compiler. The major code smells are mostly about to conditions that always evaluate to true, empty code blocks and useless assignments. The minor smells are almost all about unnecessary casts and usage of line continuation. These code smells could be refactored.

Lastly, the most significant source of technical debt according to SonarQube is the cyclomatic complexity. The most complex file is the type checker: checker.ts. This file has a cyclomatic complexity of just over nine thousand, making the file very unreadable. However, this file is well tested and the developers do mention in their coding guidelines that they don't want to split up any classes [24]. This adheres to the Single Responsibility principle from the SOLID design principles. However, it could still be argued that this class could be split up since there are different stages of type checking. This would increase the readability of the type checker of the compiler.

6.2 Todo Comments

Next, we analyzed signs of technical debt in todo comments. Todo comments a good indicator for technical debt as they are easily searched for and provide clear information if they are written well. Furthermore, todo comments are written by developers for developers, to indicate what still has to be changed in further versions to improve the product.

In the source code of TypeScript, there are 116 `TODO` comments (measured on March 14, 2018). In Figure 6.2 the division of todo comments across the different layers (as discussed in Section 4.1) can be seen.

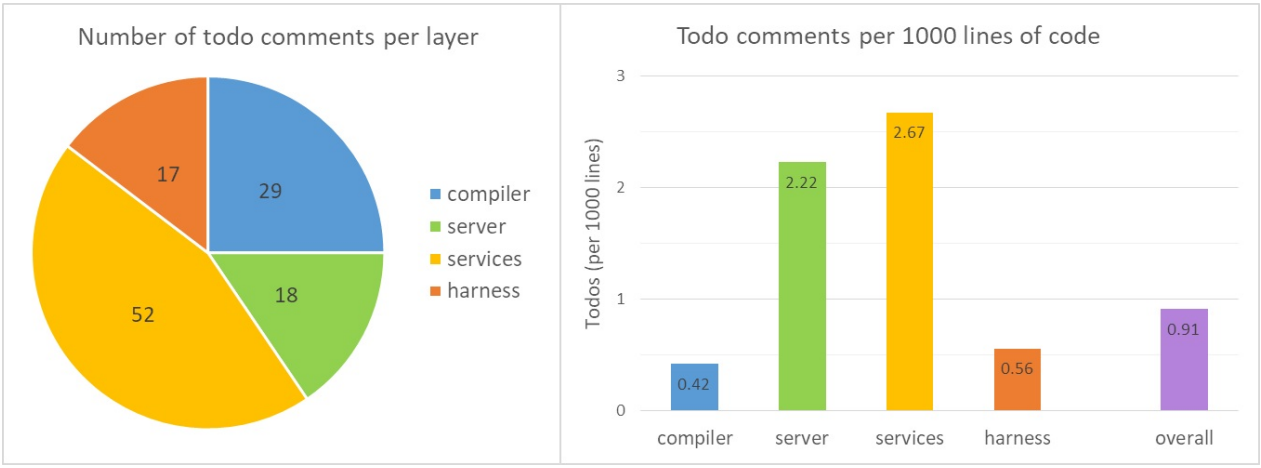


Figure 6.2 - *The image on the left shows the number of todo comments in each layer. The image on the right shows the number of todo comments per 1000 lines of code for each of the different layers.*

From Figure 6.2, we can make two remarks on the todo comments on the code. First, the layer with the most comments is the services layer. This is not surprising as this layer offers functionalities such as formatting, code fixes and debugging, which all have edge cases that only a few users stumble into. This lowers the priority of fixing these todo comments.

Secondly, while it would seem that the compiler layer has quite a few todo comments, it actually has the lowest density amongst the different layers. This is probably because the compiler layer is the core of the TypeScript project and thus it is important to keep this part clear of any technical debt.

For a large project like TypeScript, the number of todo comments is relatively low. There is less than one todo comment for every 1100 lines of code. Furthermore, many todo comments are still relevant. However, since TypeScript heavily uses the issue system of GitHub, many todo comments can be converted into GitHub issues and then be removed from the code. This would probably provide a better

overview than hiding them in the code.

6.3 Testing Debt

TypeScript uses Istanbul [25] to measure code coverage. We have generated two coverage reports for TypeScript, which are shown in Figures 6.3 and 6.4.

In both reports, any module in `src/harness` can be ignored, since this module contains the testing harness as explained in Section 4.1.

93.16% Statements 71952/77239 **88.63%** Branches 39987/45115 **90.62%** Functions 11371/12548 **93.22%** Lines 67796/72727

File		Statements		Branches		Functions		Lines	
src/harness/parallel/		11.64%	64/550	4.65%	10/215	10.45%	7/67	11.05%	58/525
src/harness/		78.69%	4992/6344	66.4%	1761/2652	76.47%	1095/1432	78.46%	4480/5710
src/server/typingsInstaller/		85.96%	202/235	73.13%	98/134	96.15%	25/26	86.3%	189/219
src/server/		89.06%	3857/4331	77.06%	1411/1831	88.39%	830/939	89%	3601/4046
src/compiler/transformers/module/		92.38%	1031/1116	88.36%	554/627	91.41%	117/128	92.24%	998/1082
src/services/		93.39%	8002/8568	89.14%	5461/6126	91.09%	1359/1492	93.53%	7419/7932
src/services/codefixes/		93.99%	1892/2013	82.76%	1123/1357	98.05%	403/411	94.64%	1767/1867
src/compiler/		95.07%	35179/37003	92.19%	24937/27050	91.01%	4647/5106	95.02%	33020/34750
src/services/refactors/		95.83%	828/864	90.7%	605/667	100%	79/79	96.03%	799/832
src/compiler/transformers/		96.27%	4494/4668	91.2%	2416/2649	98.67%	593/601	96.3%	4427/4597
src/services/formatting/		98.06%	1214/1238	93.77%	903/963	100%	212/212	97.96%	1155/1179
src/harness/unittests/		98.89%	9673/9782	83.86%	696/830	97.27%	1816/1867	98.92%	9381/9483
src/harness/unittests/services/		99.43%	524/527	85.71%	12/14	100%	188/188	99.41%	502/505

Figure 6.3 - *Coverage overview as generated by Istanbul [25].*

In Figure 6.3, for every module in the source code, the statement coverage, branch coverage, function coverage, and line coverage is shown.

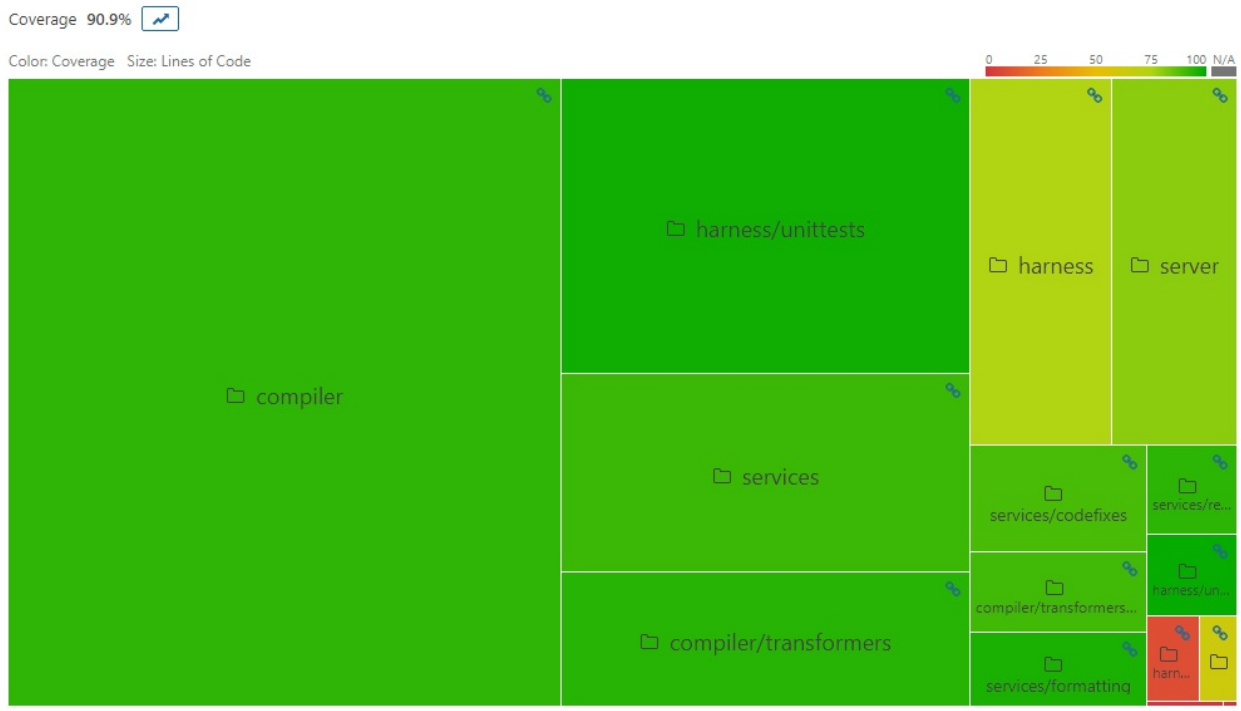


Figure 6.4 - *Coverage overview as generated by SonarQube [22] from the remapped report.*

Figure 6.4 shows the coverage proportional to the number of lines of code in a module. The size of each block is proportional to the number of lines of code, while the colour of each block indicates the test coverage (an average of line and branch coverage).

From both Figure 6.3 and 6.4, we can see that in general, the modules that are relatively small have low coverage and vice versa. Also, TypeScript has a rule that states that all changed code in pull requests must be 100% covered [26]. From this, we can conclude that testing debt is almost absent in TypeScript.

6.4 Evolution of Technical Debt

Besides the technical debt that is currently present in TypeScript, we also looked at the evolution of the system. For this we created evolution matrices [27] for the three main modules, using Matplotlib [28] and Gitcovery [29]. These matrices show different metrics for each file over time, therefore allowing us to distinguish patterns in the evolution of TypeScript. In this section, we will highlight a few interesting observations.

6.4.1 Red Giants

Red giants are components that are very large and maintain this size over many versions. The most obvious component in TypeScript that qualifies is the type-checker, with over 27,000 lines of code and steadily growing over time, as can be seen in the bottom row of Figure 6.5. Another component that acted as a red giant was the emitter, with around 8,000 lines of code, shown in the top row of Figure 6.5. This component is also a case where we can see a successful refactoring (in version 2.1). Here, its size was more than halved, making it more maintainable.

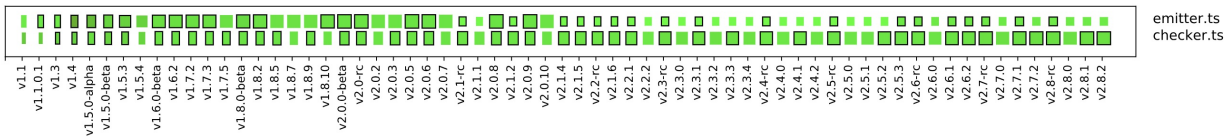
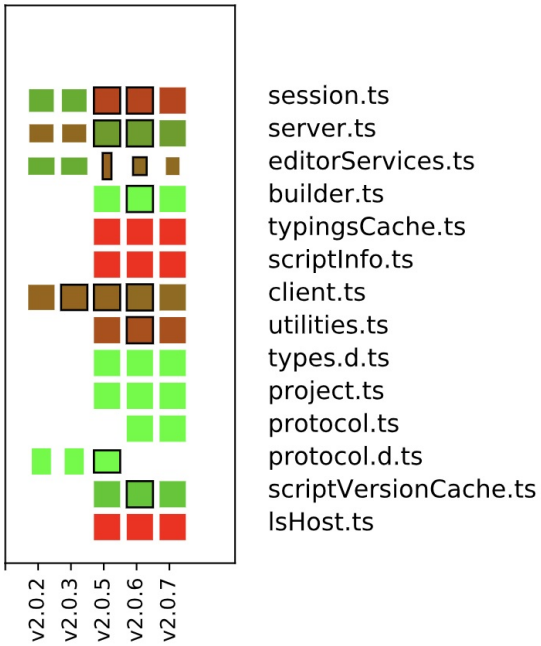


Figure 6.5 - *A subsection of the full evolution matrix, showing the emitter and the type-checker in more detail. The width and height of the rectangles show the number of functions and number of lines per function respectively (as a metric of file size). A black outline is added when a file is changed. We also normalized the sizes of the rectangles to the maximum values encountered for a component, to make the figure more readable.*

6.4.2 System Growth

During the lifespan of TypeScript, there have been multiple phases in where the number of components grew significantly. A good example is the server module, which doubled in size from version 2.0.3 to 2.0.5 (Figure 6.6a). According to the release notes, this is due to the integration of TypeScript with Visual Studio [30].
Another instance where the evolution of the system is shown clearly is in the early days of the services module. At this moment the module saw significant changes with the removal and addition of most components (Figure 6.6b). These were likely only design changes as they are not mentioned in the release notes.



a) - *Growth phase of the server module* **b)** - *Growth phase of the services module*

Figure 6.6 - *Subsection of the full evolution matrices, showing phases of growth. The width and height of the rectangles show the number of functions and number of lines per function respectively (as a metric of file size). The colour shows the number of todo comments per 1000 lines of code, where green is 0 and red is 5 or more. A black outline is added when a file is changed. We also normalized the sizes of the rectangles to the maximum values encountered for a component, to make the figure more readable.*

7. Conclusion

TypeScript is a well-designed system that has matured a lot since its 1.0 release. The design of the system is well-thought through and is very close to the architecture as it is currently implemented. Despite its growth, it has been able to maintain a good architecture that does not violate the SOLID principles. However, one thing we noticed is that it is very difficult to familiarize with the TypeScript code, simply because of the complexity of the project.

For a large system like TypeScript, it has very little technical debt. There are many unit tests and integration tests that together give a high coverage and automated tools find very few code problems. There are a few todo comments that indicate technical debt, but all things considered, TypeScript is very much on top of technical debt.

In terms of usability, TypeScript is doing a great job, both in the system itself as in the ecosystem around it. IDE support is good, there is a strong community and developers seem to like the language.

However, the real struggle that TypeScript has does not lie in its architecture, but rather on GitHub. The development team struggles to keep up with the number of issues and pull requests that are created every day. The initial response is quick, but the follow-up can take a long time, which can discourage users and can make integration of pull requests more difficult.

All things considered, we think that the TypeScript team is doing a great job. We are confident that TypeScript will be able to overcome the challenges ahead if they continue as they do now.

References

1	Microsoft, TypeScript Logo, *Microsoft*, 2017.
2	Microsoft, TypeScript Website, *Microsoft*, 2018.
3	Kleinschmager, Sebastian *et al.*, Do static type systems improve the maintainability of software systems? An empirical study, *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, 2012.
4	Microsoft, Friends of TypeScript, *Microsoft*, 2018.
5	Turner, Jonathan, Announcing TypeScript 1.0, *Microsoft*, 2014.
6	Shen, Tom, TypeScript Eclipse plug-in, *Eclipse*, 2013.
7	Bright, Peter, TypeScript, Microsoft's JavaScript for big applications, reaches version 2.0, *Ars Technica*, 2016.
8	Microsoft, TypeScript GitHub, *GitHub*, 2018.
9	StackOverflow, Questions tagged Typescript on StackOverflow, *StackOverflow*, 2018.
10	Microsoft, TypeScript Official Twitter, *Twitter*, 2018.
11	Microsoft, Microsoft Channel 9, *Microsoft*, 2018.
12	Microsoft, TypeScript Blog, *Microsoft*, 2018.
13	CoffeeScript, CoffeeScript: List of Languages that Compile to JavaScript, *CoffeeScript*, 2018.
14	Rozanski, N., & Woods, E., Software systems architecture: working with stakeholders using viewpoints and perspectives, *Addison-Wesley*, 2014.
15	Microsoft, TypeScript language specification, *Microsoft*, 2016.
16	Microsoft, TypeScript Wiki: Architectural Overview, *Microsoft*, 2017.
17	Microsoft, TypeScript Wiki: Writing a Language Service Plugin, *Microsoft*, 2017.
18	Microsoft, TypeScript: Compiler Options, *Microsoft*, 2018.
19	Microsoft, TypeScript: tsconfig.json, *Microsoft*, 2018.
20	Microsoft, TypeScript Wiki: Standalone Server (tsserver), *Microsoft*, 2017.
21	Stack Overflow, Stack Overflow Developer Survey 2017, *Stack Overflow*, 2017.
22	SonarQube, SonarQube main website, *SonarQube*, 2018.
23	SonarQube, SonarTS GitHub repository, *SonarQube*, 2018.
24	Microsoft, TypeScript Wiki: Coding Guidelines, *Microsoft*, 2018.
25	Krishnan Anantheswaran, Istanbul GitHub repository, *Krishnan Anantheswaran*, 2018.
26	Microsoft, TypeScript contributing.md, *Microsoft*, 2018.
27	Lanza, Michele, The evolution matrix: Recovering software evolution using software visualization techniques, *Proceedings of the 4th international workshop on principles of software evolution*, 2001.
28	Matplotlib, Matplotlib website, *Matplotlib*, 2018.
29	Chiel Bruin, Gitcovery GitHub repository, *Chiel Bruin*, 2018.
30	Microsoft, TypeScript 2.0.5 release notes, *Microsoft*, 2016.

Vue.js - The Progressive JavaScript Framework


Arthur Guijt


Philippe Louchtch


Tim Speelman


Lars Stegman

Introduction

Web applications have evolved to the point where DOM manipulation via javascript is exceedingly common, however most implementations doing so are slow. Entire lists have their representation recomputed while only a single item has changed. Or the data model and visual representation are out of sync due to not recomputing enough. In the meantime there is a ton of code performing string interpolation to generate the interface elements.

Vue.js is a library/framework for Javascript and TypeScript that uses templates or render functions, in combination with data binding to make building user interfaces easy and clean. It can be used for a small part of the website as a library, up to full single page applications utilizing Vue as a framework. Applications can also be adopt Vue incrementally, allowing developers to slowly introduce it in your project. Vue won't break out of nowhere. It is well tested: with a suite of over 1000 unit tests, practical e2e tests, and it boasts of 100% line coverage.

Vue bears many similarities to other javascript libraries/frameworks related to DOM manipulation, most notably React - as they both use a virtual DOM - but also other frameworks like Angular. For an in depth comparison we would like to refer you to this comparison, while it is not independent it gives a good overview of the differences between projects.

In short, quoting the first page of the Vue.js guide:

> Vue (...) is a progressive framework for building user interfaces. (...) Vue is designed from the ground up to be incrementally adoptable. (...) (Vue) is easy to pick up and integrate with other libraries or existing projects. (...) Vue is also perfectly capable of powering sophisticated Single-Page Applications (...).

This introductory quote neatly outlines the scope of Vue.js: handling the view layer, mainly by DOM manipulation and event binding. In contrast with jQuery, which provides an alternative API to DOM manipulation, Vue takes the entire DOM manipulation away, letting the user focus on writing logic.

Stakeholder Analysis

Core and Ecosystem Distinction

Vue.js is more than the "core" Vue library itself. We have identified a strong interplay between the core Vue library and the various components and extensions. The development of one guides the development of the other.

Therefore we have split the developers and maintainers into "core" and "ecosystem" camps. We concern ourselves with the analysis of the "core" part. However, the ecosystem cannot be ignored, hence we included it in our developer and maintainer stakeholder analysis.

Users and End-Users Distinction

The Vue.js project is not a product directly usable in the traditional sense. It is a library, middleware making it easy to build web-based client-side applications. We make a distinction between two classes of user stakeholders:

- End-users
  This is a virtual class of stakeholders of users who end up using the product (the end-product) built by the 'Users' but are not

exposed to the Vue.js project directly. They are concerned with having a compatible, fast and solid end-product and as a whole are uncaring about the specific middleware used.

- Users
  These are the direct users of the Vue.js project. They're concerned with having happy end-users and thus share their concerns. On top of that, they are the stakeholder who pick the middleware and thus care about technical aspects such as: strong community support, tooling, features, and clean and easy to use APIs.

Vue.js Community

The community is a virtual group of people participating in Vue.js development and maintenance, as well as Vue.js related discussions on GitHub & Vue.js forum, Vue.js discord server, Vue.js StackOverflow topic, Vue.js related conferences, etc. This community is mostly comprised, but not exclusive to, the Vue.js developers, maintainers, users and stakeholders whose businesses depend on the continued success of Vue.js. Please note that participation in the community is completely voluntary.

The Stakeholders

Acquirers

Evan You is the main and most important acquirer of the system. He set out to build something new and to prove its viability. The success of the project attracted sponsors, recurring and one-time donations. The whole body of the sponsors can also be considered, to a lesser extent, as an acquirer or a virtual entity with interest in the continued existence of the system.

It is unknown to us if there are hidden sponsors involved, i.e. companies whose products are built on top of Vue.js and have a direct line of communication with the development team.

Communicators

Evan You is the face of the project and is the most prolific member of the team. Being the face, he promotes and communicates. Furthermore, there are commercial trainers and an organization that organizes Vue.js conferences. Every member of the core team, some more than others, represents the product through their interaction with the community, whilst also writing documentation.

Core-Developers

Just like the 1.0 version was a large rewrite over the 0.xx series, the current 2.x series is a large rewrite over the 1.x series. The developers of the 2.0 release were the people that were members of the "core" team at that time source. An overview of the (current) core members can be found here

Core-Maintainers

The maintainers of the Vue.js core library are the members of the Vue.js GitHub organization, which is a super-set of the core developer group. The actual maintainers are the active and willing members of the Vue.js GitHub organization.

Any member of the organization has the ability to approve or decline a pull-request. Evan You, however, is the only person making the final merges and releases.

The Integrator

Evan You is the man, the face and the brains behind the Vue.js project. It is both his reputation and source of income. Evan is directly dependent on the success of the project, therefore he is concerned with keeping the relevant stakeholders happy. These include: the users (and indirectly the end-users), developers and maintainers, and sponsors. Furthermore, the community size and richness of the ecosystem is a strong indicator of interest, therefore his interest lies in keeping stakeholders happy whilst growing the community and the ecosystem.

This is a special kind of stakeholder, much like Linus Torvalds of the Linux Kernel fame, whose success is directly related to the success of his most renowned project. Hence, in a respectful, tongue-in-cheek kind of way, we originally referred to this stakeholder as "The Linus". Another interesting aspect of this kind of stakeholder is the "Benevolent Dictator" development/Git-branching model where only the integration-manager (or "the dictator") has direct commit privileges to the "blessed repository".

Ecosystem-Developers & Maintainers

Some members of the core-team are also ecosystem developers. However, the ecosystem is not a specific project or a repository. There are both "official" and community ecosystem contributions. The official Vue.js components live directly in the Vue.js GitHub organization, while the community components start their life outside of it.

Users

The direct users of the Vue.js project. These are the developers or companies making use of Vue.js to build their own products.

One very important user of Vue.js is the Alibaba Group. Not only have they built some of their websites with Vue.js, they decided to help integrate Vue.js with Weex in persuit of their goal to use only one front-end framework across their applications, both web and mobile. source

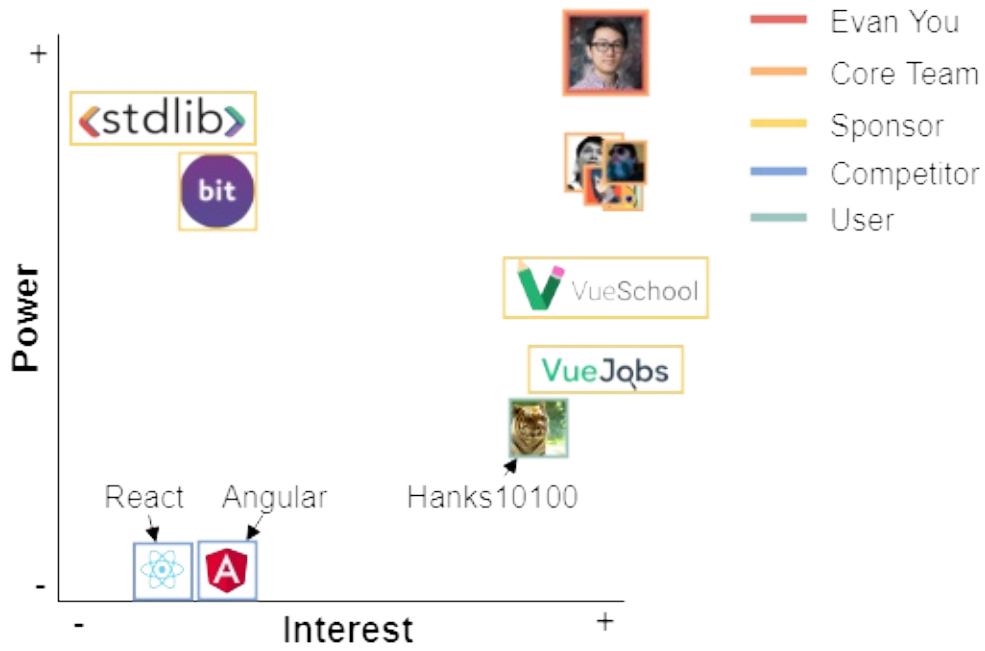
End-users

The users of the final product made by the "Vue.js users". This group does not participate in the development of Vue.js but their concerns are extremely important for the evolution of the Vue.js project. Their wishes and demands may propagate up to the Vue.js ecosystem or even the core project.

Support Staff

Some communicators also fulfil this role by participating in stack-overflow and forum questions & .discussions

Power-Interest



This power interest diagram shows the most important stakeholders, below is an explanation for each of them.

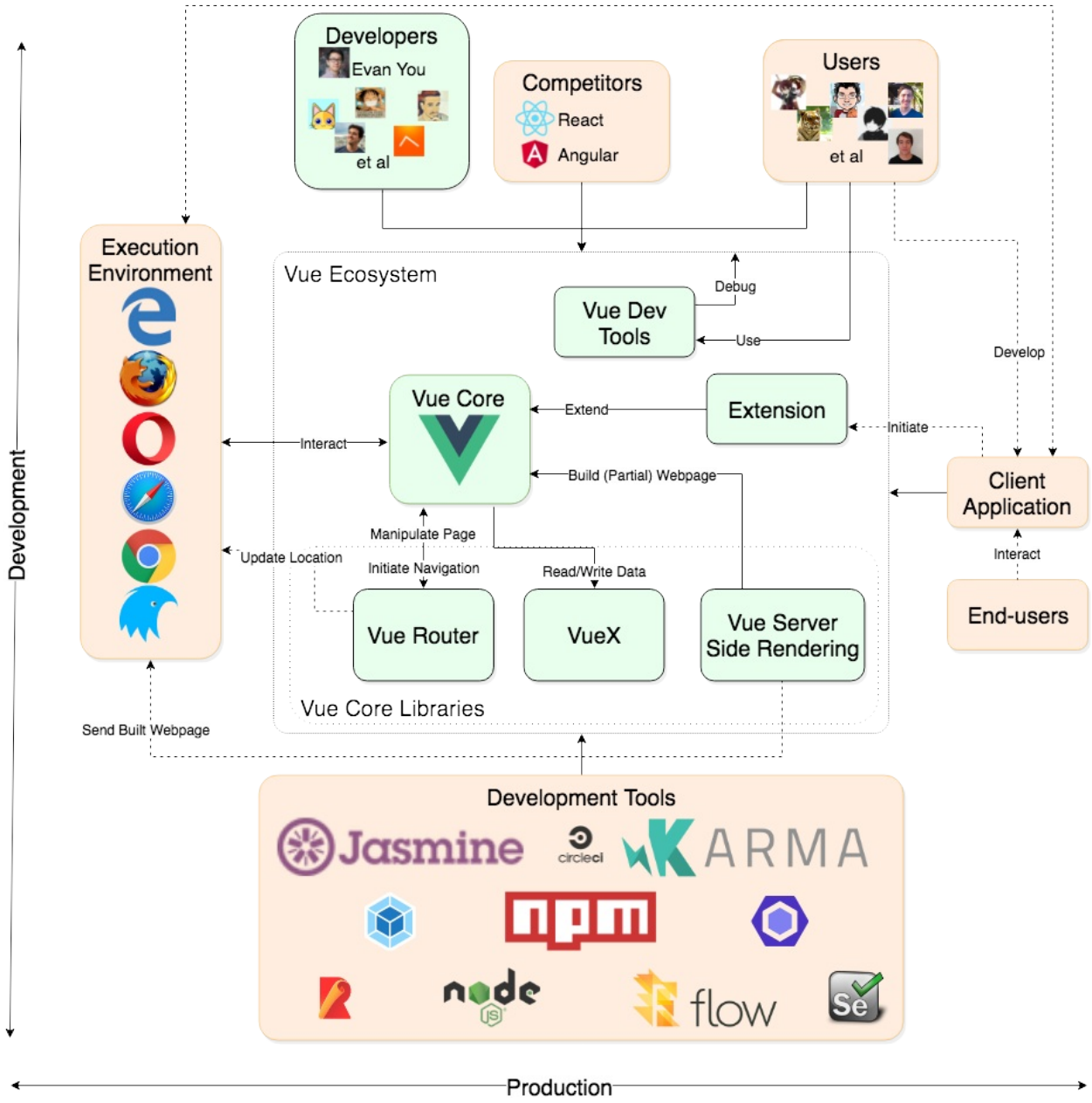
Evan You is by far the most important person in the system, and is the one who makes the decisions. Due to Evan being the final decision maker and *the* integrator, the core developers have less power. Furthermore, their interest is rated lower as their income and reputation does not depend as much on the success of the project as Evan does.

The top-tier sponsors contribute significant amounts of funding to the project. Such funding always buys some level of power, even if subconscious, in the case of Vue we argue this also being true, albeit limited to some extent. Not all sponsors are directly dependent on the success of the Vue.js project, these score lower on the interest graph. Others like VueSchool, do and therefore are rated having higher interest in the success of the project.

Hanks10100 is an example GitHub user who is not part of the development team. We consider him to be part of "the Vue.js community".

Finally we have two major competitors, React and Angular. While they do not hold much power, as competing products they have to pay attention to each other. Case-in-point: some of the people involved with React joined the discussion in the comparison.

Context view



horizontal axis: production line, vertical axis: development line

A few external entities can immediately be identified: Vue can be extended through **components and plugins**, consumed by **web applications** or **individual UI components**. Together these form the interface towards the **end user**, who uses a **browser** to run it all. This is what we call the 'production line'.

Conversely, different people and projects are involved during development which are not seen during production itself. The **Vue developers**, the **users** that create their applications, the **competitors** that can alternatively be chosen from. But also the **development tools** themselves: npm for package management, rollup and webpack for bundling, and the testing tools further described in Standardization of Testing. This is what we call the 'development line', visualized vertically in the diagram.

The part of Vue that we are looking at - Vue 'core' - is the base library on which the rest of the Vue ecosystem is built. The ecosystem consists of libraries, some of which where important enough to be maintained by Vue core developers: the Vue core libraries. Examples of this are Vue Router and VueX.

Vue is based around reusable components, some of which will be local to their own projects - but some are reusable enough that they can be used across projects. The community provides a large amount of these reusables, and many of those are shared as open source, for example those listed on awesome-vue.

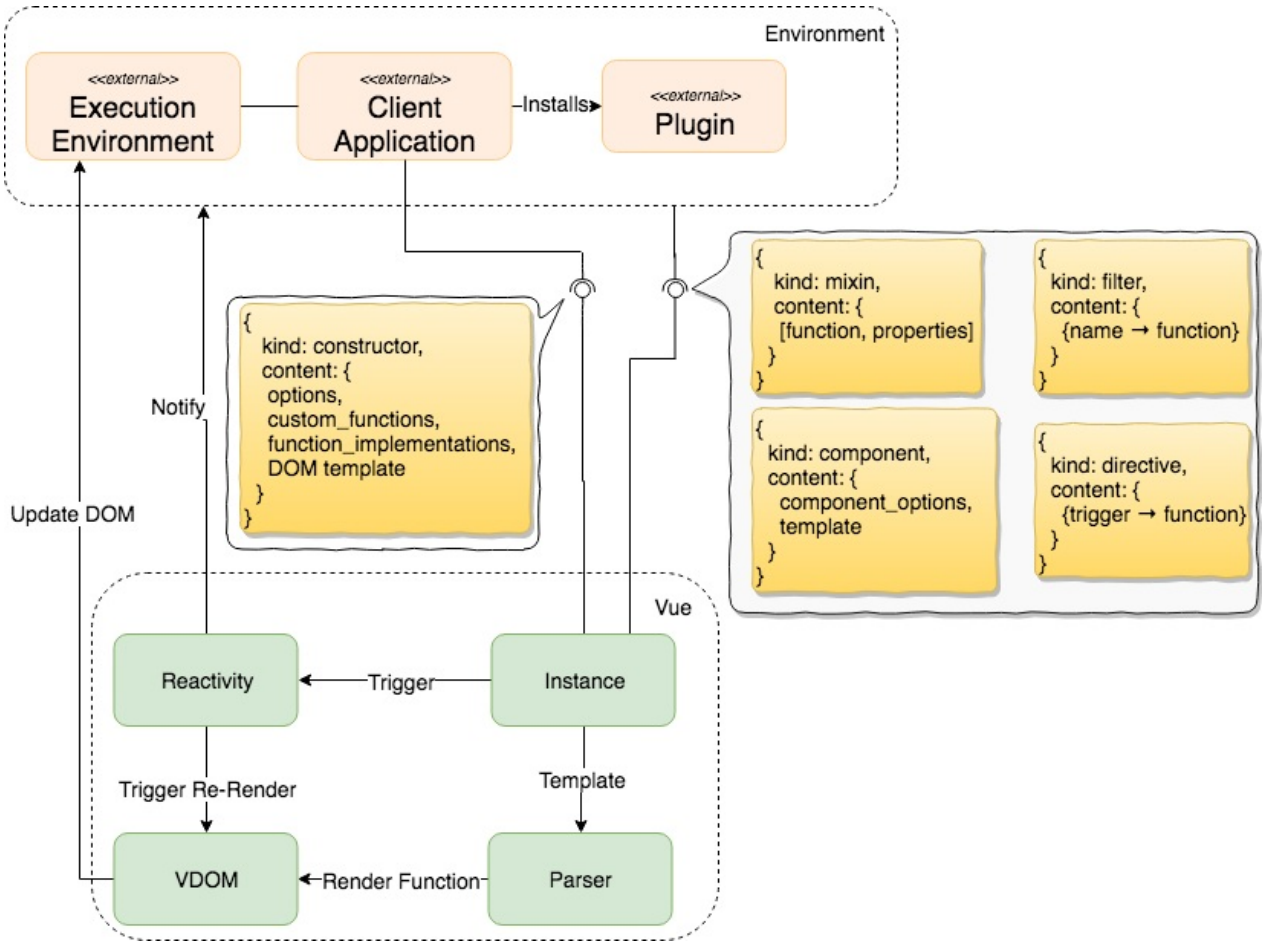
Functional Architecture

It's in the name. Vue offers a convenient way to manage the view layer. This entails two ways of communication: the data from the application is presented to the end-user, and the end-user input is communicated back to the application. These two are essentially the main 'visible' functionalities Vue offers.

Furthermore, Vue offers its users, i.e. web developers, two more non-functional functionalities: component-based design and reusability management.

In this section we first discuss what responsibilities Vue has and how it works internally. After that we explain how Vue communicates with its environment.

In the figure below an overview is given of the APIs Vue provides and what the internal components send to each other.



The Internal Architecture and the Functionalities It Offers

Everything the end-user can interact with, is the product of applying a format to the application data and sending changes back to this data when the end-user changes something.

Formatting

The recommended way of specifying a format in Vue is using the Vue template syntax: HTML with three sugars, namely custom tags, custom attributes and interpolation (e.g. `{{text}}`).

In short, a Vue app defines a set of templates and passes these to Vue along with the data it wishes to display. Vue's parser then parses the provided template, turning this entire sugared custom HTML string into a JavaScript render function; a function that takes data and outputs a Virtual DOM Node (or VNode), ready to be injected into the real DOM tree. This process is displayed in the diagram above.

Reactivity

This is definitely not enough: for a one-off render one might as well use PHP. Instead, whenever the data changes, Vue will update the view accordingly using its reactivity system. To monitor changes, the system attaches itself to the application data object. As a full re-render on every data change would be too costly, Vue pulls a few tricks to make efficient changes to the DOM tree. The backbone of this is the dependency tracking system. It only updates those parts of the tree that depend on changed data (e.g. `{{text}}` only re-renders when the value of `text` changes).

Input handling

Every interaction the end-user has with the application is a product of the application methods (API) and some bound UI elements. In the view template it is possible to connect events of view elements to methods. This is mainly handled by the `v-on` directive, a custom html attribute provided by Vue. Similar to the data formatting, Vue parses the template and binds the provided method to the actual event. On top of that, it will destroy this 'listener' when the component is removed from the DOM.

Component Re-use

Vue is entirely component based. Any use of Vue requires instantiating it using `new Vue(options)`, which creates a Vue instance. The instance takes a template as mentioned above, some data and some methods and Vue handles the rest. Larger apps will want to separate and re-use logic instead of writing one big component. For this, one can define a Vue component using `Vue.component('my-comp', options)` and subsequently use it in Vue templates as `<my-comp />`. This facilitates separation of concerns and code re-use. Vue's third main functionality is offering a way to define components and subsequently handling their entire: from its creation, through several updates of its state, to it finally being destroyed. Larger applications, extensively using Vue, will have to shape their architecture towards it. So in return, Vue must consider how it can best facilitate architectural best practices to these applications.

Interactions With The Environment

Vue provides a diverse set of external interfaces, from lifecycle hooks to complete plugins. This section describes each external interface and how they can be used by users to extend Vue's behavior and functionality. First, the lifecycle hooks are explained. Second, the use of components, directives, and filter, i.e. assets. The concept of mixins is then discussed, and finally, plugins that can be used to encapsulate logical modules.

Lifecycle Hooks

Vue allows users to observe the lifecycles of Vue instances by implementing functions that are lifecycle hooks. Users can use these hooks to perform certain tasks at certain points in time during the instance's lifecycle, e.g. start fetching some data from an external server. For a detailed explanation of all existing lifecycle hooks please see Vue's user documentation.

Assets

In addition to adding custom behavior to Vue instances, users can also add custom assets to Vue. These assets can be used in templates to build a DOM.

Component

Components are the building blocks of the (virtual) DOM. They have their own internal logic, data structures and reactive components. Components are identified by their names, for instance `counting-button`, would be a fitting name for a button that counts the number of times it has been clicked.

Directives

Directives are dynamic properties that add behavior to components. Directives are defined using the `Vue.directive` method. An example of a directive is the default directive `v-show` which toggles the visibility of a component in the DOM depending on the arguments passed to the directive.

It is also possible to create custom directives that add behavior to the elements they are added to. Examples can be found in Vue's documentation. The most interesting example is the custom directive `focus`, which automatically focusses elements when they are inserted in the DOM.

Directives can respond to various events from the elements they are applied to. A complete list of the events can be found here.

Filters

The final asset type that is discussed are filters. Filters can be used to format data, for example transforming a Unix epoch to a human readable timestamp. Filters are defined using the `Vue.filter` method.

Mixins

Mixins allow users to add custom properties and functions to Vue instances.

For example, Vue Router adds the properties `$route` and `$router` to Vue instances, which respectively represent the current route the end user is at and the router object the user can use to control routing.

Vue Router also uses mixins to implement the lifecycle hooks `beforeCreate` and `destroyed` to be able to dynamically update the current route the end user is at.

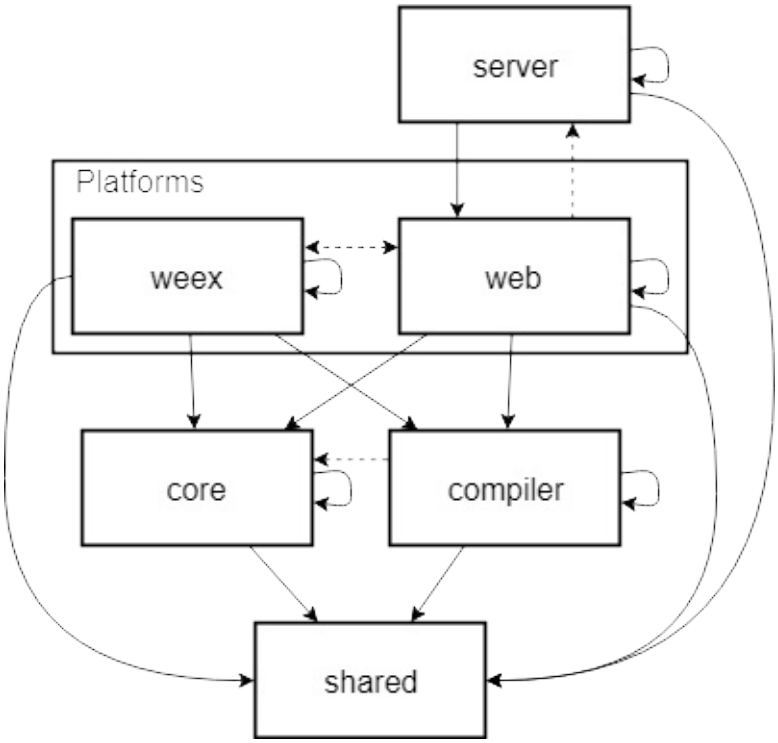
Plugins

Adding custom behavior in an encapsulated manner is also possible with Vue. Completely separate plugins can be created and then plugged into Vue through their `install` functions. This function is used by Vue to instantiate the plugin and allows the plugin to add custom behavior to the Vue instance.

Usually, plugins use the mixin and asset concepts described above to add their behavior to Vue instances. For examples of this we recommend looking at the `install` functions of Vue Router and VueX.

Development View

Module Organization



Vue consists of a few modules: The shared module contains common utilities that are not included in the standard javascript library. The core on the other hand contains logic specifically related to Vue. This logic is split up in submodules, some of which are:

- `vdom` -- virtual DOM manipulation module
- `observer` -- provides internal functionality to make certain constructs *observable*, a design pattern. To allow Vue.js to *react* to model and/or state changes in a relatively (to their competitors) performance efficient way.
- `global-api` -- the general global api that is provided to the users.
- `instance` -- Lifecycle management, events and state of Vue are located here. This module is required to run in environments which does and does not have the web APIs related to manipulating webpages.

In order to make things fast certain parts of the user-provided templates can be parsed, optimized and compiled beforehand by the compiler. The entry points are located in three modules. The Weex module is for identically named framework developed by Alibaba, web is for running in a webbrowser related context, while server is meant for server side rendering. These points contain the platform specific code to let Vue run and work within their respective environments.

Isolated from the rest of the codebase is the module that contains the parsing logic for single file components - a feature that allows Vue to be used as a framework. This part is used by plugins for build tools - for example webpack and rollup - to deal with single file components - without having to re-implement the parser.

Standardization of Design

In order to streamline contributions from the community, Vue provides a contribution guideline document. The document explains how to set up Vue for development, report issues, creating PRs, and an explanation on the project folder structure, among other things.

Surprisingly, the project structure is the only specification of design choices made for the Vue core package. Moreover, it is the only real source of documentation on Vue source code. The lack of comments in the source code is noticeable, and gives developers a hard time understanding its workings and motivations behind it.

In sharp contrast to the core package itself, Vue docs offer an extensive developer style guide for its users. This gives developers who use Vue strong advice in terms of naming, code style, use of particular Vue constructs (such as using a `key` prop when using the `v-for` directive) and handling of data. As these guidelines do not apply to the core, it is beyond the scope of this text.

Standardization of Testing

The Vue.js tests are located in the `/test/` directory. This directory contains various types of tests for the core Vue.js library. Because the core Vue.js library also contains platform specific code, tests for those platforms are also included. Two of the platforms are quite specific and have their own unit-tests: Server-Side Rendering (SSR) and Weex. The other, generic, browser platform is tested through the end-to-end tests using Selenium. These tests are located in the `/test/e2e` directory and are used to test the correctness of Vue.js behavior on the currently most dominant browsers.

Core Vue.js Tests

Located in the `/test/unit` directory and can be further categorized into unit and integration tests. Structurally, these are split into `module` and `feature` types.

The `/test/unit/feature` Tests

Tests to ensure the correct operation of user-facing Vue.js features and global API. These are, for the most part, integration tests.

The `/test/unit/modules`

Tests to ensure the correct operation of the underlying building blocks of core Vue.js library itself. Namely the `vdom`, `compiler` and `observer` submodules.

Platform-specific Tests

The rest of the directories (except for `helpers/`) deal with testing the various platform-specific code that is part of the core Vue.js library. There are three logical platforms:

- Server-Side Rendering (SSR)
- Weex
- Web

The SSR and Weex platforms are special kinds of platforms. These are platforms which are used in non-standard environments. SSR renders templates into HTML by running the client-side code on the server before serving. Weex is a platform for shipping Vue.js-powered (amongst others) applications as mobile, native applications.

Both Weex and SSR have their own unit tests that deal with platform specific features, behavior, and past quirks and bugs.

The web platform is the default Vue.js platform, namely the major browsers. These cannot be tested with JavaScript unit tests and therefore are tested with end-to-end tests and interaction scenarios. Interesting is that the test-data used for the End-to-End tests are the live-examples of Vue.js, although some specific test-data is also present amongst the Selenium test scenario definitions in the `/test/unit/e2e/specs` directory.

Technology Used

All JavaScript tests are written using the Jasmine JavaScript unit testing library and Karma JavaScript test runner.

The end-to-end testing is done with Nightwatch.js which uses the Java-based Selenium internally to simulate user interaction. Nightwatch.js simplifies writing technical end-to-end tests for Selenium and is script-able with JavaScript.

Testing Pipeline

Due to the distributed and parallel development nature of the project, where any person can propose a contribution, keeping the code quality high requires some work from the core team.

This quality control is partially implemented in the form of an automated testing pipeline. The pipeline is implemented as an npm script and consists of the following steps:

1. ESLint
2. Type-check with Flow
3. Unit tests with coverage

4. End-to-end tests

source

This testing pipeline is also executed by the project's cloud based CI-tool on each commit. A failing pull-request will not be approved.

Approaches

With regard to testing approaches, there is no guideline other than a brief mention in the contributor guide of new changes needing to have "appropriate test coverage if applicable".

There is some standardization of some common actions in the `/test/helpers` directory. These deal with either setting up some test case or with helping the developer to do some common assertion.

Technical Debt

As most applications, Vue has some technical debt. This section analyzes how serious Vue's technical debt is and where it is located.

Code Smells

A good indicator of places where technical debt might be located is the presence of code smells.

Lack of Documentation

One of the first code smells we noticed was the lack of documentation; a very consistent lack of documentation. Especially complex pieces of the code, such as the `compiler`, would benefit from adding documentation. This has two benefits: obviously, others can understand what is going on, but maybe more importantly, explicit mentions of the reasoning can prevent bugs when refactoring or otherwise modifying the code later.

SOLID Principles

Another way of identifying technical debt is by looking for violation of the SOLID principles. However, Vue is written in JavaScript, object-oriented programming is not enforced and barely used within Vue. The SOLID principles are harder to apply, but may still be useful.

The Single Responsibility Principle (SRP) seems to be well respected. Most code is split up into well-named files and functions, clearly identifying their single responsibility. One exception to this is a number of `utils` files, spread across packages. While sometimes justified, these files are a magnet to technical debt.

The Open-Closed (OCP) principle may loosely translate to the frequency with which files are changed. The more frequent a file is changed, the more likely it is that its functionality is changed instead of extended through subclassing, and thus that OCP is violated.

The other SOLID principles are less applicable to Vue.

Other Smells

There were two other code smells that stood out.

The high cyclomatic complexity in some files. The violations of this were `server/render.js` (108), `compiler/codegen/index.js` (125) and `compiler/parser/index.js` (138). Which makes this even worse is there was almost no documentation in these files, making it even harder to understand how the code works.

The second smell that stood out was the method length. There were methods that were hundreds of lines long. This smell mainly occurs in high performance code, which makes it more justifiable, but this should be kept under control. One of the worst violators of this was the filter parser. We have made a contribution to fix this problem to reduce complexity and method length.

Even though Vue has ESLint configured, it is not used to its full potential. The issues of long methods was not detected by it and other syntactic issues that we found, we also not detected.

Testing

Vue.js has an extensive test suite and has a whopping 100% line coverage according to the continuous integration. This was surprising as hardly any project can achieve 100% coverage, which is why we looked deeper into the testing practices of the Vue.js project.

Continuous Integration

Vue has CI enabled on all pull requests and branches on GitHub. This encourages contributors to properly test their code before submitting it, as nobody wants to break the build.

The Vue tests are written using Jasmine. This framework allows testers to test asynchronous code with assertions and spies. Vue has multiple kinds of tests: unit, end-to-end, Weex, SSR, type, and Sauce. The Sauce tests are equal to the unit tests with the exception of the execution environment. Aside from Sauce, all tests are executed on the CI server.

Coverage

In addition to running the tests on every pull request, the CI also measures coverage. According to CI the coverage of Vue is 100%, but after analyzing the codebase, we found annotations that disable coverage for certain parts of the code. After we removed these annotations we saw that coverage actually 95.34%. The majority of uncovered code is present in the HTML parser, which is actually an essential component of Vue. It is however not custom written code, and a clone and own strategy has been applied to it.

Testing Practices

The tests for Vue are written using Jasmine. Jasmine allows developers to test asynchronous code using assertions and spies. The tests for Vue are written quite well and are often easy to understand.

They could be improved even more by making sure that tests do not test more than one functionality at the same time. A couple of tests test more than one thing, which makes it harder to pin down what goes wrong when the test fails.

Another thing that could improved is the "unitness" of the tests. When we worked on our contribution and broke one component, over 70% of all tests began to fail.

Testing Environment

The end-to-end tests for Vue are currently being executed using PhantomJS, however PhantomJS has some problems. The main developer has quit and the last minor release - as of writing - is from January 2016 which, given the velocity regarding WebAPIs and the JavaScript language, can be problematic.

We recommend the Vue team to replace PhantomJS with Chrome Headless. We have created a pull request in the Vue repository to remove some barriers that exist to adopt Chrome Headless.

Development History

Over the course of its existence, Vue has had multiple rewrites. We have analyzed the codebase over its releases and in this section we discuss the results.

From the analysis we were able to see that there is a positive correlation between the amount of changes between releases and the cyclomatic complexity in the files. This was not entirely unexpected as the files with the highest complexity are the parser and the virtual DOM files. These files are likely to change between releases, as they need to support new features.

These files are also the least readable files in the project, which might indicate technical debt. It could, however, also be caused by handwritten optimizations as these files are performance critical.

Conclusion

Vue has some technical debt, however at the moment it seems to be manageable and managed. The main points that should be kept under control are the parsers and the virtual DOM patching. One point that should definitely be improved to encourage contributions would be to improve the developer documentation, as it is nearly non-existent.

XMage: Magic Another Game Engine

By Christiaan van Orle, Siyu Chen, Danny Plenge and Marc Zwart

Abstract

XMage is an open source Magic the Gathering client and game engine that allows you to play Magic the Gathering games online against human or computer players with full rules enforcement. The core developers get help from many contributors to keep the engine up to date with new card set releases. XMage's architecture is analyzed from multiple perspectives and viewpoints. An overview of the architecture is provided, as well as an analysis of the technical debt present in the system.

Table of Contents

Introduction

XMage [5] is a project which started development in early 2010. Its goal is to make a free, open-source, online playable computer game version of the original Magic the Gathering card game with actual rule enforcement, something which competitors do not have. The project is written in Java and was initially developed by *BetaSteward*, who still moderates the online forums of the XMage project. Nowadays, it is in the hands of *LevelX2*.

The chapter starts with performing a stakeholder analysis and a context view of XMage. After that, the module organization and the codeline organization are presented in the development view section. The technical debt is analyzed from different points of view, followed by the deployment view, where the requirements to run the system are demonstrated. Additionally, the project is discussed separately from regulation perspective and usability perspective. Finally, a conclusion ends the whole chapter.

Stakeholders

This section presents an overview of the types of stakeholders found in the XMage project. Besides the Stakeholders as defined by Rozanski and Woods [1] we have also Identified Competitors and Non-sourcecode Contributors. We could not identify other stakeholder types than the mentioned ones. This diagram is a general overview of all the important stakeholder categories in the project. Per stakeholder category, including the additional stakeholder categories we found ourselves, it shows all the relevant stakeholders. Figure 1 below is a more detailed description of all the stakeholder categories. Next, the Power/Interest grid and the Context view based on the stakeholder analysis will be presented.

Figure 1: *Stakeholder Overview*

Power vs Interest Grid

This section introduces a power/interest grid based on the stakeholder types we have found and analyzed, as shown in Figure 2, each stakeholder type is represented by a MTG card [2]. The power and interest of each type of stakeholder is based on the pull request analysis, issue analysis and the analysis of the forum, Gitter and Reddit.

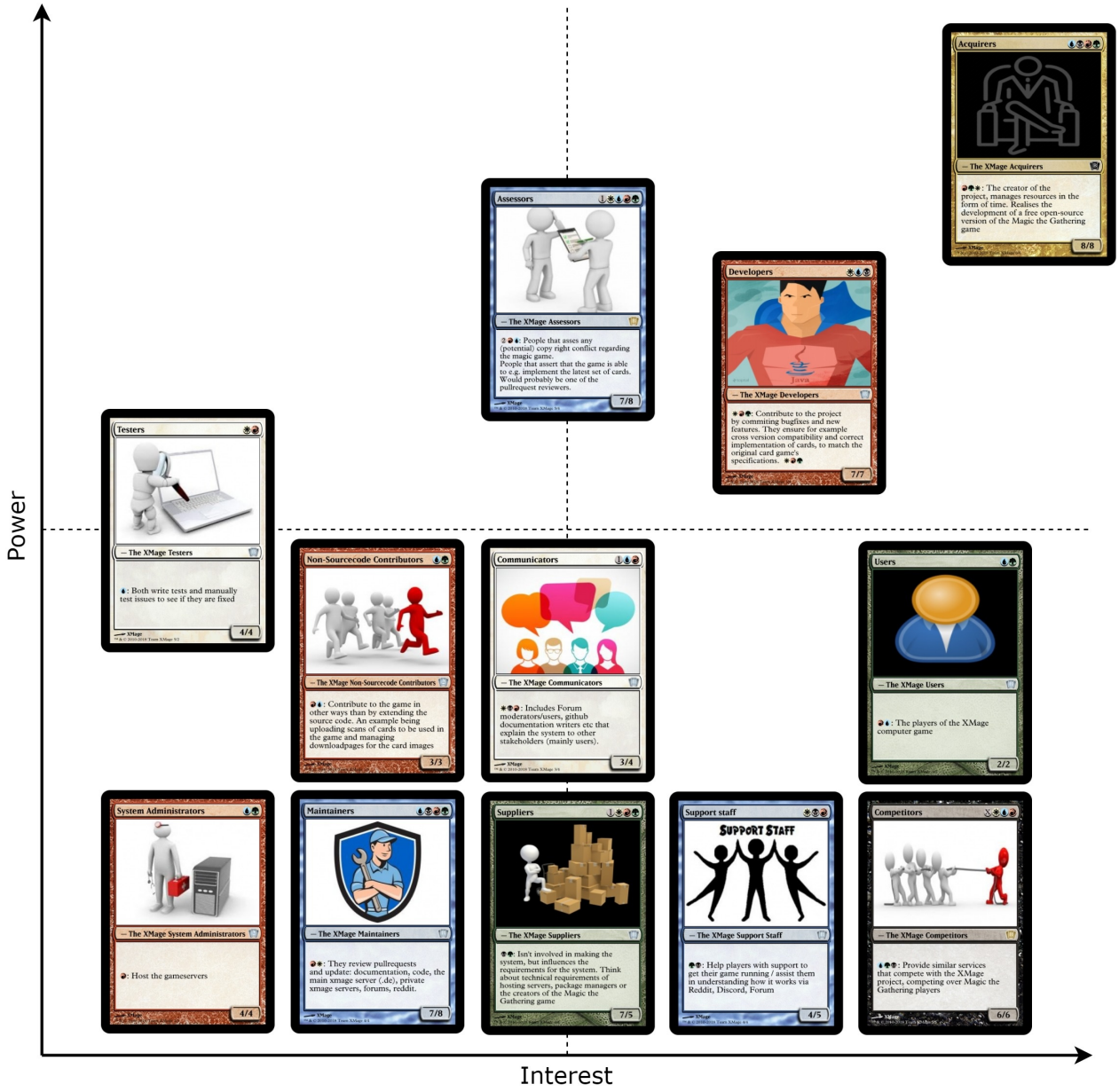


Figure 2: *Power vs. Interest Grid*

Context View

The Context View contains the relationships, dependencies and interactions of the system with the environment, as shown in Figure 3. It shows the communications channels used in order to communicate, the most important stakeholders, the dependencies of the system and the operating systems the system supports.

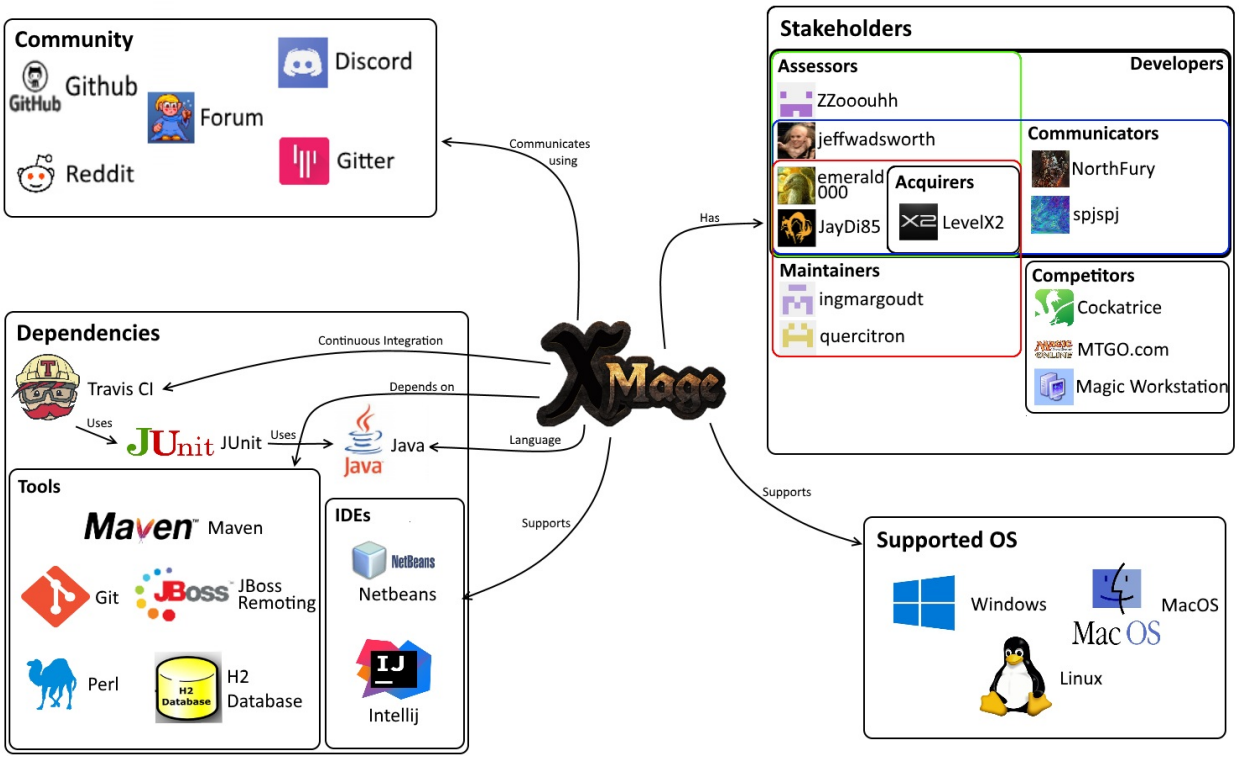


Figure 3: *Context View Diagram*

External Entities

This section touches the external entities of the XMage project, we will elaborate on the different dependencies and the communication platform the community uses. As the project is written in Java, all Operating Systems can run the XMage game.

Community

The XMage project largely lives around its community, the users, developers, communicators etc. communicate with each other via several different platforms. Users among each other tend to use the Reddit pages, users may also post help requests on either the Reddit or Forum to find communicators that can help them resolve their issue. Developers communicate with each other via the Gitters and Github. The Discord channel is used for finding other users and is even used for tournament communication.

Dependencies

The XMage project has several dependencies of different types. The entire project is a Maven project written in Java which consists out of different Java projects. The project can be build using any Java supporting IDE, however at the moment there is only support for the Netbeans and IntelliJ IDEs. The continuous integration is done by Travis which uses JUnit for testing. The client server communication is handled by JBoss Remoting 2. In order to provide quick access to card data a H2 database is used for the client, server and test project. The project also has some utility scripts written in Perl which can help the developer generate code or build certain projects.

Development View

In this section we will start by looking at the XMage project [3] at module level, identifying module organization and design patterns. Some light will also be shed on the codeline organization, further discussing source code organization and the build approach.

Module Organization

In this section we will look at the organization of the XMage project from a Module perspective. The XMage project is separated into two major parts, the client and the server. Besides this a validation step is included in the build process which forms a relevant part of the project.

Client-Server Architecture

Here we will discuss the client-server architecture used in the system, as well as what modules are used in which component.

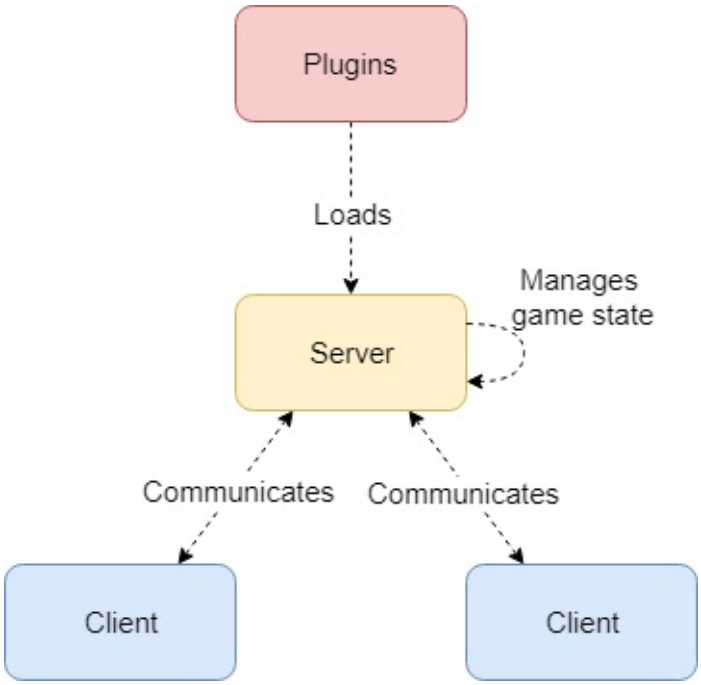


Figure 4: *A High Level Overview Architecture in the XMage Project*

Structure

In XMage, both client and server are running as separate instances of the game. Each instance communicates with the other via sockets, this communication layer for both client and server is implemented in the *Mage.Common* module. The server manages the game's state and updates the clients, where clients send their respective actions to the server. Both the client and the server implement the core game logic by depending on the Mage module. A high-level overview of the main (runtime) components of the software is shown in Figure 4.

The overview in Figure 5 shows the modules present in the system. Some modules live in both client and server programs, such as the *Mage* module. The validity assessment program is not present on either server or client, it is not used on production systems at all. The *Mage.Server* and *Mage.Client* modules are responsible for running the client and server programs. They load the required resources, such as the cards and game logic (plugins).

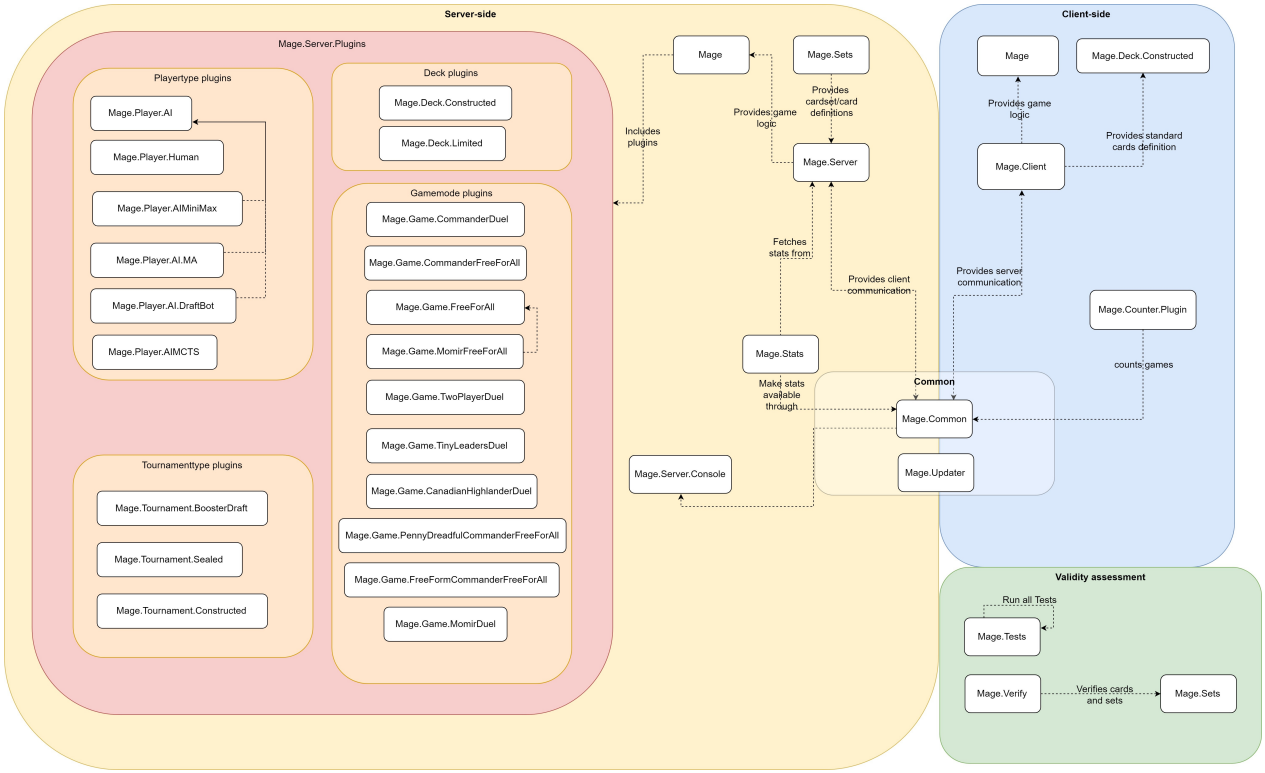


Figure 5: *A Detailed Overview of the Module Organization in the XMage Project*

Server Plugins

The server supports plugins and plenty of them are available by default from the XMage project. The Mage module can load different types of plugins, each of whose interfaces are exposed from this module. The supported types of plugins are listed as shown below in Table 6.

Plugin Type	Description
Player	Allow implementation of player types, examples are the Human player (controller by keyboard/mouse) and the AI (controlled by java code)
Deck	Allow definition of different deck types, examples are a predefined deck (used in some tests) and a limited deck (ensures a minimum amount of cards)
Tournament	Allow the definition of tournament types, comprising of various tournament steps and rules
Game mode	Allow the implementation of game types, from simple two player duel's to more complex game modes like Commander free for all

Table 6: *The Supported Types of Plugins in the Server*

The Mage module dynamically loads all available plugins, however this only happens on the server, as the client gets updated by the server it does not need to load the plugins itself. It may be clear that this, lower level, architecture implemented is a plugin-architecture.

The module organization of plugins is shown in Figure 5.

Validity Assessment

A third interesting part of the system is the validity assessment which happens during the build, verifies all implemented cards and runs all the tests. Cards are extensively validated to ensure that no duplicate or invalid card-names/-numbers/-types etc. exist. This allows the developers to assess that the thousands of cards are in a valid state before releasing a new version of the game.

Pattern Usage

Several different design patterns that have been used throughout the XMage project to aid in making the system extendable and maintainable. First and most easy to spot are the huge amount of Singletons [6] used for object for which only a single instance is required. For several types a serialization-safe implementation for these has been used as they all are enums instead of classes to prevent singleton duplication after deserialization. A lot of Ability classes throughout the project are singletons too.
We also see a variety of Factories [7] implemented, loading or creating various objects. The most notable ones are the factories that create the respective objects for each type of plugin. For example there is a PlayerFactory which loads the Player type plugins and exposes methods for constructing Player objects based on these Player types. A second example is the GameFactory for loading Game types and exposing methods for constructing Match objects based on these Game types.

Codeline Organization

In this section, we will look into the code level organization of the XMage project. First, we will present the structure of source code, and then the build will be discussed.

Source Code Structure

Both files and folders are found in the repository [3]. In Table 7, the files under the root directory are listed with descriptions.

File	Description
read.md	A documentation including general information of the project, the installation and developer's instructions
pom.xml	Configuration file for the project
clean_dbs.sh	A script to clean cards.h2* from Server, Client and Test modules
.travis.yml	Configuration file for setting up Travis CI
.gitignore	Configuration file for git describing which files should not be sent to the remote repository

Table 7: *All the Files under the Root Directory*

The repository of XMage is organized based on the modules, as shown in Table 8. Except for the last folder 'Utils', each folder represents a component in the system. This codeline model was chosen based on the fact that the system contains multiple components. It ensures that the project will be developed in a clean and well-organized way. It also provides clear clues for the developers where to check problems or develop a new feature. Some of the source code folders in the modules also contains testing code besides the main code. All the folders are described in the following table.

Folder	Description
Mage	The logic of the game
Mage.Client	The client side with Swing UI, which displays game states and sends player events to the server
Mage.Common	The communication of the elements shared between the client and the server
Mage.Plugins	A counter plugin to display the number of how many games were played
Mage.Server	RMI server which maintains tables and games, sends updates to clients and receives client events
Mage.Server.Console	The console of the server
Mage.Server.Plugins	All the plugins of the server
Mage.Sets	All the card sets and cards
Mage.Stats	The stats of the server
Mage.Tests	Automatic tests for XMage
Mage.Updater	Updating the system based on metadata from remote server
Mage.Verify	Asserting correctness of card definitions in *Mage.Sets*
Utils	Perl scripts for developing and updating cards and card sets

Table 8: *All the Folders under the Root Directory*

The Build Approach

In general, XMage is built on five main modules, including the game logic, the cards and the card sets, the server, the client and the communication between the server and the client [8]. The system was extended by applying more plugins for the server and adding more modules with different purposes. The build standard for contributing by adding new cards and new sets is described in details on Developer HOWTO Guides [4] page. Travis CI, a continuous integration platform, is used in building the whole project. Every time when there is a new commit or a pull request, Travis CI will build it and then deploy to Heroku or update the PR. Travis CI ensures the quality of the source code, and also helps implement the system quickly and detect errors easily.

Technical Debt Analysis

This section discusses some technical debt analysis from different points of view. We will start high level by looking at the responsibility separation between modules, then go more in depth by analyzing the code metrics we have found from a SonarQube scan. Finally, we will discuss some of the code we have seen in the project during manual analysis.

Internal Dependencies

In the mage project we found a rather clean separation of responsibilities of modules. Especially the separation of state (Mage.Server), logic (*Mage.Framework*) and view logic (*Mage.Client*) is a nice design for the game. Separating the Client/Server communication into a separate module (*Mage.Common*) and defining the Card sets and Cards into a separate module while depending on the Framework containing the abilities/spells etc. also contributes to a solid design.
A poorer design choice we have found in the *Mage.Server* are the dependencies on each plugin. Because of this, the Server cannot compile without these dependencies, analyzing the actual implementation this does not seem like a necessary dependency. Each plugin can dynamically get loaded by the server through a jar file (which can be referenced from a config file) therefore making the dependency redundant.

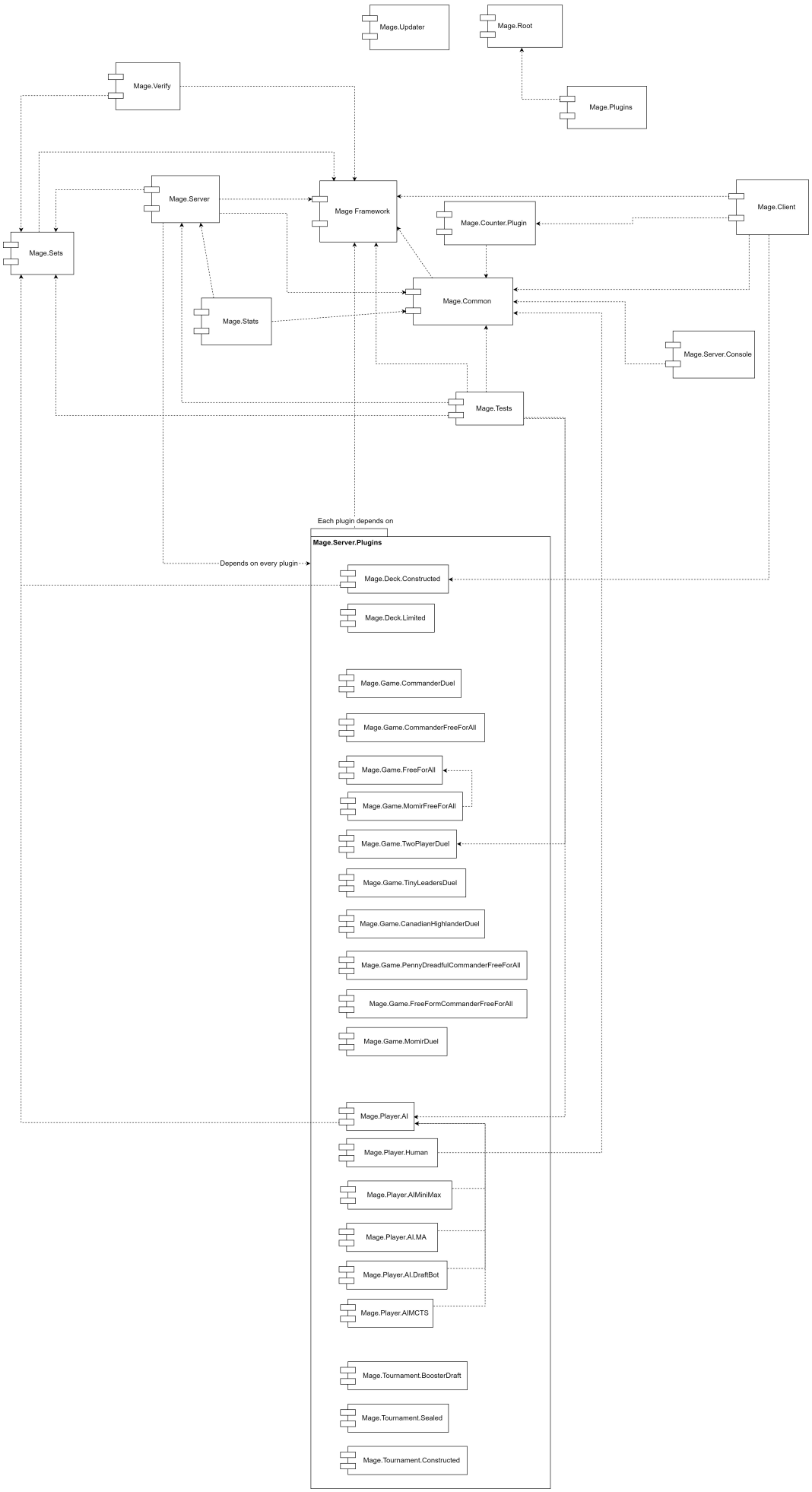

Mage.Updater
Mage.Root
Mage.Verify
Mage.Plugins
Mage.Server
Mage Framework
Mage.Client
Mage.Counter.Plugin
Mage.Sets
Mage.Common
Mage.Stats
Mage.Server.Console
Mage.Tests
Each plugin depends on
Mage.Server.Plugins
Depends on every plugin
Mage.Deck.Constructed
Mage.Deck.Limited
Mage.Game.CommanderDuel
Mage.Game.CommanderFreeForAll
Mage.Game.FreeForAll
Mage.Game.MomirFreeForAll
Mage.Game.TwoPlayerDuel
Mage.Game.TinyLeadersDuel
Mage.Game.CanadianHighlanderDuel
Mage.Game.PennyDreadfulCommanderFreeForAll
Mage.Game.FreeFormCommanderFreeForAll
Mage.Game.MomirDuel
Mage.Player.AI
Mage.Player.Human
Mage.Player.AIMiniMax
Mage.Player.AI.MA
Mage.Player.AI.DraftBot
Mage.Player.AIMCTS
Mage.Tournament.BoosterDraft
Mage.Tournament.Sealed
Mage.Tournament.Constructed

Figure 9: *A Dependency Graph of all Mage Modules*

Code Metric Analysis

From our SonarQube scan we have analyzed some of the metrics of the project. While the project consists of over a million lines of code as of the latest release (1.4.28) we see that 80% of that belongs to the *Mage.Sets* module, second being the *Mage.Framework* making up about 12% of the project in lines of code. The *Mage.Sets* module is by far the biggest module in the project and this reflects in other size metrics as well like Lines of Code (LOC), amount of Statements [13], amount of Functions or Classes.

Metric	Total	Mage.Sets	Mage.Client	Mage.Server	Mage.Framework
LOC	1,000,000	800,000	50,000	10,000	125,000
Statements	400,000	275,000	25,000	5,000	50,000
Functions	100,000	80,000	3,000	800	15,000
Classes	26,000	23,000	400	80	2,500
Code Smells	14,000	7,000	2,500	300	2.500
Cyclomatic Complexity	150.000	100,000	10,000	2,000	25.000
Bugs	300	20	180	25	35
Vulnerabilities	400	30	175	5	50
Duplication	2.4%	2.2%	3.5%	1.1%	2.8%

Table 10: *Various Metrics for the Main Modules of XMage (Rounded for Clarity)*

Although the Mage.Sets module is, as seen in Table 10, the largest module, surprisingly enough it does not contain most bugs and vulnerabilities found by SonarQube. We see high amount of Bugs and Vulnerabilities found in the Mage.Client project, indicating poorer code quality than in the rest of the project. The very low amount of Bugs and Vulnerabilities in other modules indicate a good code quality. Seeing a total of 700 Bugs and Vulnerabilities together on a million lines of code project seems rather good. The project contains low amounts duplication, again given its size this is a good indication. From this analysis we conclude that the overall quality of the code is good, the amount of bugs is reasonable given the size of the project. Regarding improvements we think the Mage.Client module needs the most work, ridding it of the bugs that were found and decreasing the complexity of the overall code. This last point also considers the Mage.Server and Mage.Framework modules, as their complexity with respect to their size in LOC is roughly the same.

Code Quality

The static code analysis indicated good overall code quality, but we still found some bad practices in the implementation of XMage. Throughout the code we see large numbers of nested control flow statements, some with depths that make the code near impossible to understand. We consider improving these blocks for better understandability of the code.

Furthermore, we have noticed a violation of a SOLID principle, namely the Dependency inversion principle. We have depicted a part of the XMage project's class hierarchy in Figure 11. Here we see a good example of the Dependency inversion principle, namely the CardImpl abstract class with children implementing this abstract class. Throughout the code CardImpl is used to depend on any implementation of CardImpl, as the Dependency inversion principle dictates. Now we also see the Token class, which is a concrete class and is used throughout the code for both itself and its children. This results in a dependency on a concrete type which we would rather see implemented like the Cards. The impact of this is that extending the Token class may cause confusion for developers, causing misusage of the Token class or not using references to the correct token class.

Our suggestion to fix this is to make Token an abstract class and rename it to TokenImpl in order match the naming conventions in the XMage project. Then replacing any concrete usages of Token by a suitable implementation of TokenImpl, while leaving all dependencies on TokenImpl. This would result in the same situation that is present for the CardImpl. Besides resulting in fixing the SOLID violation this will also yield higher consistency in the project.

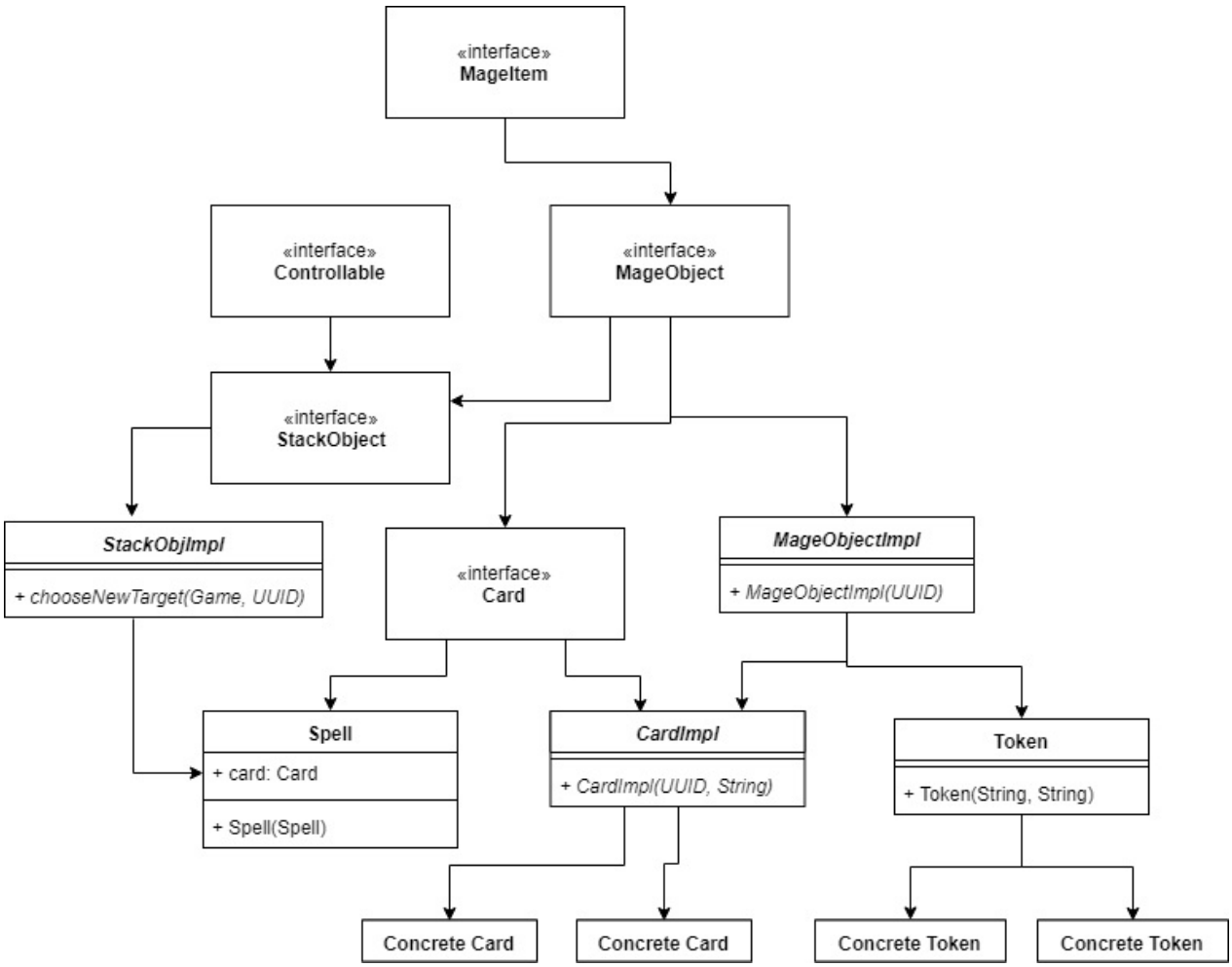


Figure 11: *A Part of the Mage Class Hierarchy*

Testing Debt

This section discusses the testing debt of the XMage project. This includes the test structure, the test coverage, how manually testing works and the test procedures of the XMage project.

The Test Structure

The XMage project exists out of many modules. Some of these modules are only made for testing the project like the *Mage.Tests* and the *Mage.Verify* modules. The *Mage.Tests* module consists of tests which cover multiple parts of the project like the server, client and AI but most of all the Mage Framework. These tests are there to make sure that the current functionality of the program does not change by any code changes. The Mage Verify module is more focused on testing the implementation of new cards. It checks all kinds of conditions like whether there are no duplicate cards or whether the card class names are correctly written.

Testing Procedures

In the XMage project it is not required to add tests for each new implemented card. However, if you make any changes in the *Mage.Framework* module it is required to properly test your code. Especially if you make any additions it is required to add the necessary tests to make sure your implementation doesn't break anything. This is why the Mage Framework is tested mostly by unit tests, where all the cards are mostly tested manually. When you do add a test, it is important that you add your test in the *Mage.Tests* module, that it is a JUnit test and that it properly builds in the Maven project.

Discussions about Technical Debt

In this section, we dive into the XMage Github to search for the discussion between the developers on the technical debt. First, the open issues are analyzed in terms of technical debt, and then we discuss the search results of TODO's [16] and FIXME's [17].

The developers of the XMage project did discuss some of their technical debt in Github issues [14]. Most of the issues related to the technical debt were raised 2 to 3 years ago, which are about the programming logic between some objects, the game logic, the cards management and the source code organization. However, some of developers who discovered the issues or joined the discussions are no longer active in the project, and the others decided to keep the current solutions. Recently, they have discussed about not implementing the illegal cards to prevent technical debt, since the illegal cards ask for the abilities that the system does not support [15].

Based on the fact that there are numerous outdated TODO's and FIXME's, it is highly suggested that an issue should be raised to discuss them. For example, for the ones which are still valid a schedule should be arranged and related developers should be assigned to solve them, the ones which do not affect the system anymore should be removed. Additionally, another issue should be created to schedule all the recently-added TODO's and FIXME's.

Evolution of Technical Debt

This section discusses the evolution of the technical debt in the system. First, the code base will be discussed, then the use of TODOs and finally the impact of the technical debt will be shortly described.

Code Base

The system has evolved a lot over the past 8 years. At the start of the project, there were around 30000 lines of code. At the end of 2017, there were over a million lines of code in the system. It appears the code base has increased very consistently over the years. The SonarQube analysis shows that in the past year the number of lines of code, files and classes coincide with the increase in technical debt.

TODO Analysis

The number of TODOs increase steadily over time. As can be seen in Figure 12, the growth in the number of TODO's coincides with the growth in lines of code.

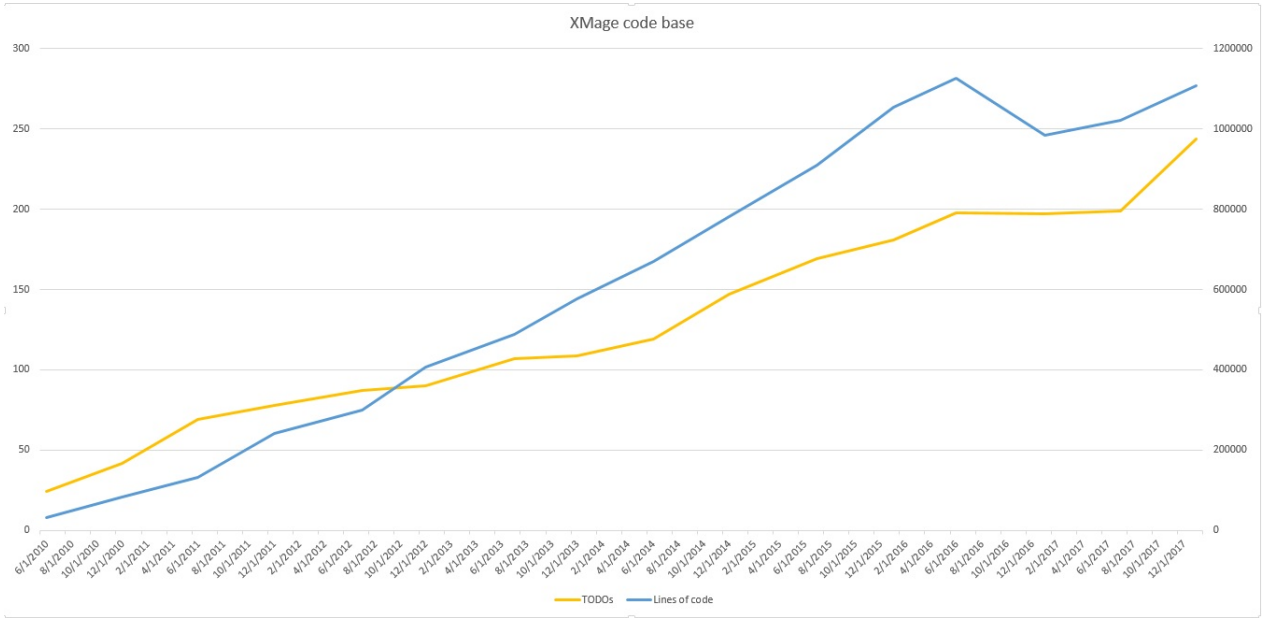


Figure 12: *A Graph Showing TODOs versus Lines of Code over Time*

It may seem like the number of TODOs increase as more code is written, and the TODO/lines-of-code has been decreasing over time. Further analysis shows however, that most of the time, TODOs are actually ignored. As can be seen in Figure 13, there are TODOs that have existed for over 7 years.

Figure 13: *A TODO in a piece of code, which was created 7 years ago.*

Looking at the TODO message, it appears no one ever looked back at this piece of code. A large number of very old (5 years or older) TODOs are found in the classes related to Artificial Intelligence (AI). It appears the difficulty of making the AI better was too high, and efforts to improve it were abandoned.

Impact of Technical Debt over Time

As can be seen in Figure 14, the technical debt appears to have an impact on the number of bugs in the system.

Figure 14: *A Graph Showing the Technical Debt versus the Number of Bugs*

As the technical debt increases, so does the number of bugs find in the system by the SonarQube analysis.

Deployment View

This section discusses the hardware, networking and third-party requirements. Even though there are not that many requirements to run XMage, there are still some important requirements to meet. The overview of deployment view in the XMage project is shown in Figure15.

Figure 15: *The Overview of Deployment View in the XMage Project*

Third-party Software Requirements

Both the client and server require Java 1.8.x to be installed.

System Requirements

As both client and server run on Java, XMage can run on many operating systems. XMage has been tested to work on Windows, Linux and Mac.

Client

The client requires about 200MB of disk space to run. Optionally, card images can be downloaded, requiring 10GB of disk space. 512MB of memory is required to run the client.

Server

The popularity of the server determines the memory requirements. For the most popular server from a few years ago [9], this was about 3GB of memory for the server, resulting in about 1GB per 100 users, excluding operating system memory usage. The server will require about 150MB of disk space to run. The CPU usage is not very high, so any CPU should be enough to serve a large number of players.

Server Networking Requirements

Networking may be difficult. There are a few things that have to be considered.

Network Setup

The server has to be accessible by the intended users. For public servers, this means allowing access to ports 17171 and 17172. If the server is behind a NAT router, these ports will have to be forwarded to the server.

Network Capacity

The average upload speed required is about 10 Mbit/s for 250 users, while the download speed is about 10% of the required upload speed, as can be seen in Figure 16.

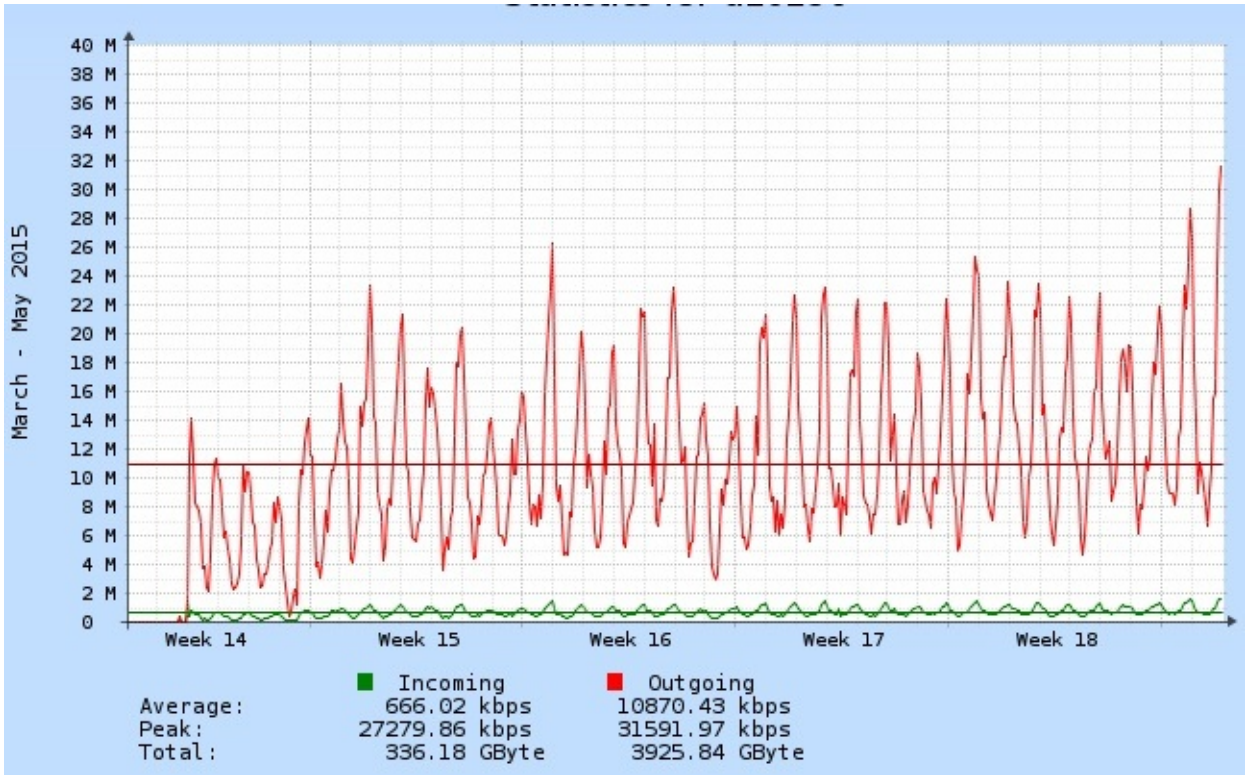


Figure 16: *The Bandwidth of Network in the XMage Project*

Runtime

Java has to be downloaded to run the launcher, but the launcher will then download a specific version of Java to ensure all users run the game using the same version.

Regulation Perspective

The XMage project is mainly affected by the copyright surrounding the Magic The Gathering game. In order to circumvent the legal issues coming with adding the images and packing them with the game, they were left out of the XMage project. Instead a download feature was implemented from which the images could be downloaded from an external source, leaving XMage out of the way of any

potential copyright infringement. Solutions less pressing on the players of the game were suggested and discussed but like all discussions [10] they came down to the copyright infringement problem.

Open Source License

When going through the project a lot of the files contain the: *"Copyright 2011 BetaSteward_at_googlemail.com. All rights reserved."* license in the comments, where also a lot of the files have no license reference. *BetaSteward* has not worked on the project for years, where the current active team of the XMage project does not deal with any licenses. In our perspective a license, even for open-source projects, is an important part of a project. A license should be added to the project in order to grant new collaborates the permission to use and change and improve the code any way they want, without anyone being able to claim ownership nor make any author liable for any damage his work may have done. Setting up an license is free and should not event take that much time.

Usability Perspective

In this section explains how the users interact with the system and which user interacts with which part of the system. Then we go into the depth of the different kind of user capabilities.

Users and Capabilities

We identify two main types of users in the XMage program, the players and the server administrators. The players use the game through the game's Graphical User Interface (GUI), its interaction we will describe in a followup section. From this we can already see that there are no strong technical capabilities required of the players, since their entire interaction with the game happens through a GUI. It is however expected that they know the MTG game, as there is no explanation regarding the rules of the card game embedded in the computer game. The server administrator have to do some more low level interactions [12] like manually editing xml config files, setting up cron jobs [11] and creating script files. They can manage which games are going on on the server and which players are present via a special console. It may be clear that there are more significant technical capabilities required of the server administrators.

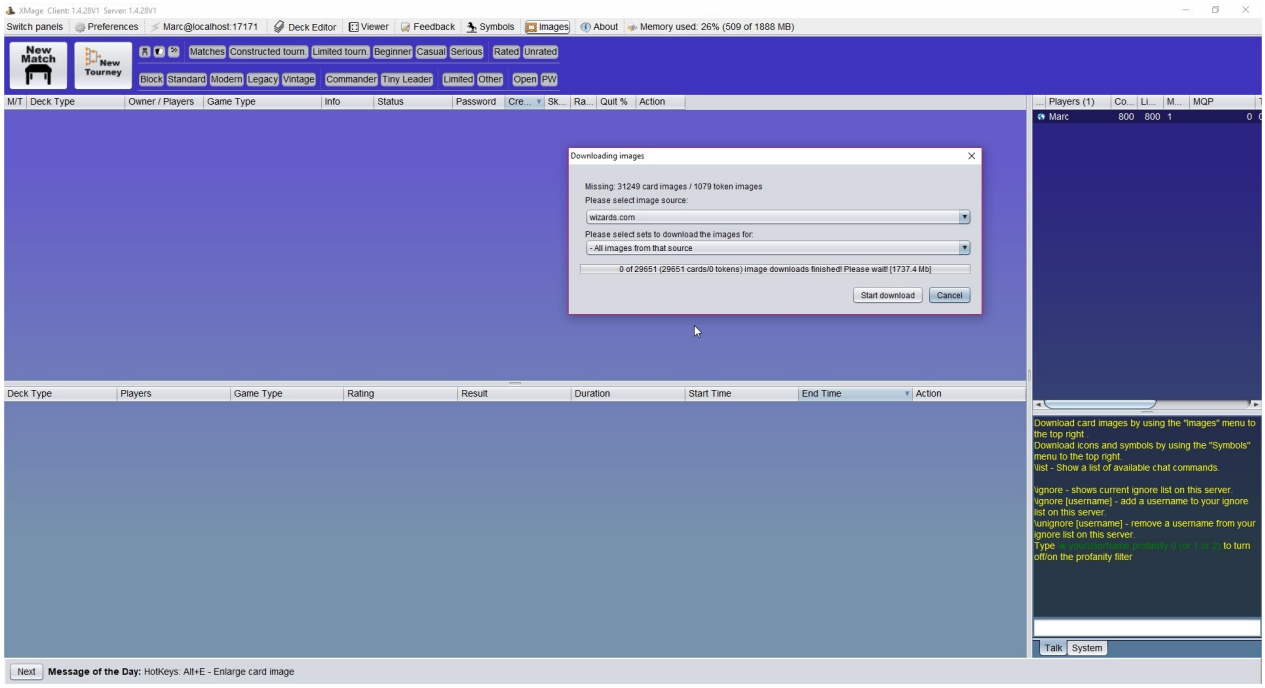


Figure 17 *The main menu of XMage, with the image download menu opened.*

Server Administrator Interaction

The server administrator first has to set up the server, this is done by following the procedure described in [12]. In this procedure the server will be configured by various options and the available plugins can also be defined. The logger is configured and cron jobs are defined to regularly restart the server. So far all interaction happens via terminals and file editors. Once the server is running there is of course the option to shut it down/restart it via the terminal, but besides that there is only one way to interact with the running server instance, via the Mage Server Console. Via this console an Admin can see which players are connected to the server, which games are currently being played and by whom. Players can be send messages to, they can also be disconnected or banned via this window.

Player Interaction

In order to make use of the XMage project the user has to start up the XMage client to interact with the user interface. If the user wants to participate in magic the gathering games he has to join a XMage server. At the moment XMage provides several servers which any user can join after creating an account. When the user has logged into the server he/she is able to start a MTG game or tournament and the user will also be able to configure any relevant properties. From this screen the user is also able to chat with the other users on the server.
The user interface also provides several other options like the preference menu where the user can change all kind of properties of the interface of the XMage client. The user can also interact with the system by creating or modifying his/her MTG decks by clicking on the deck editor tab and in order to view the actual cards themselves the user can click on the Viewer tab. Because of regulation issues the client does not provide the image and symbol resources from the start, however the user can download these resources themselves by using the symbols and images tabs in the user interface.
These are the most common ways the default user interacts with the system in order to make use of the XMage project.

Conclusion

XMage is an open-source project which has been worked on for the past 8 years. During these 8 years it has grown from a project with 30000 lines of code to a project with over a million lines of code. Because the project has had many contributors over the years, the code base has grown into a complex structure with many modules and dependencies. In this chapter, we have analyzed the architecture of the XMage project by presenting the *Server-Client* structure and explain about the many plugins it uses. We analyzed the test structure of the code including the test module and the validity assessment. We analyzed the many stakeholders the project consists of and discussed the technical debt of the project including the testing debt, code quality and code metrics. XMage has proven to be a strong competitor in the Magic the gathering scene. It has become a very interesting free alternative for online magic the gathering and has lots of potential to keep on growing. We do recommend adding an open-source license to the project and improve their merging strategy to make XMage more suitable for the future.

Bibliography

[1] Rozanski, N. Woods, E. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012. https://www.viewpoints-and-perspectives.info.

[2] MTG Card Maker. https://www.mtgcardmaker.com/.

[3] The XMage project. Magic Another Game Engine. https://github.com/magefree/mage.

[4] The XMage project. Developer HOWTO Guides. https://github.com/magefree/mage/wiki/Developer-HOWTO-Guides

[5] XMage. http://xmage.de

[6] McDonough, J. E. (2017). Singleton Design Pattern. In Object-Oriented Design with ABAP (pp. 137-145). Apress, Berkeley, CA.

[7] Hannemann, J., & Kiczales, G. (2002, November). Design pattern implementation in Java and AspectJ. In ACM Sigplan Notices (Vol. 37, No. 11, pp. 161-173). ACM.

[8] The XMage project. Developer Notes. https://github.com/magefree/mage/wiki/Developer-Notes.

[9] The XMage Project. Server load / user disconnects - what does cause the problems #662. https://github.com/magefree/mage/issues/662#issuecomment-68996043

[10] The XMage Project. Card images not bundled? #2698. https://github.com/magefree/mage/issues/2698

[11] The Open Group Base Specifications Issue 7, 2018 edition. http://pubs.opengroup.org/onlinepubs/9699919799/utilities/crontab.html

[12] Collectible Card Games Headquarters. How to set up a (public) XMage server. https://www.slightlymagic.net/forum/viewtopic.php?f=70&t=15898

[13] SonarQube. Metrics - Statements. https://docs.sonarqube.org/display/SONAR/Metrics+-+Statements

[14] The XMage Project. Issues. https://github.com/magefree/mage/issues

[15] The XMage Project. Issues. UST - Card Implementation Tracker (Tracking issue for Unstable) #4233 https://github.com/magefree/mage/issues/4233

[16] The XMage Project. Search: todo. https://github.com/magefree/mage/search?p=14&q=todo&type=Code&utf8=%E2%9C%93

[17] The XMage Project. Search: fixme. https://github.com/magefree/mage/search?utf8=%E2%9C%93&q=fixme&type=Code

Contributions

All teams put substantial effort in contributing to the projects under study. Not in all cases the proposed changes were actually merged. Many teams started with simple documentation changes to get used to the change process, and then proceeded with more ambitious changes.

This chapter lists the changes that were approved or merged into the projects under study.

Akka

- Merged pull request 24655: Add log() method to typed Logging API, fixing issue 24648.
- Merged pull request 24850: Add tests for BoundedBlockingQueue.
- Merged pull request 25025: Fix BoundedBlockingQueueSpec against spurious wakeups fixingfailing test in 24991

Angular

- Merged commit 7c45db3 for pull request 22285: docs: correct grammar mistakes in CONTRIBUTING.md

Docker

- Merged pull request 2537: Fix possible data race in manager/state/store/memory_test.go
- Merged pull request 2589: Add company and date to license
- Under review: Pull request 2595: Upgrade containerd version and update containerd executor
- Reported issue 2563: Make script fails on go version go1.10 darwin/amd64

Eden

- Merged pull request 1480: Update Arabic and Korean translations.

ElasticSearch

- Merged pull request 29255: Updates documentation on generating test coverage reports
- Merged pull request 28905: Add a usage example of the JLH score
- Merged pull request 29348: Fix some code smells in equals methods

Electron

- Merged pul request 1170: Update CONTRIBUTING.md
- Merged pull request 295: feat: parse tutorial sidebar nav content

Godot

- Merged pull request 17243: Fix being able to create folder name with ending '.' on Windows

- Merged pull request 17865: Ctrl+Clicking a enum now scrolls down to it in the docs.

Kubernetes

- Merged pull request 1887: nidorano sig cli contribution ptr in CONTRIBUTING.md
- Merged pull request 1883: Correcting an outdated URL in contributor cheatsheet

Lighthouse

- Merged commit bfb9bb5: docs(contributing): fix link for closure annotations
- Merged pull request 4680: docs(contributing): fix link for closure annotations
- Merged pull request 4912: cli(output): Add ability to export results to CSV

LoopBack

- Merged pull request 1151: refactor(CLI Templates): move start error catch to application.main
- Merged pull request 1231: refactor: move example packages to examples folder

Mattermost

- Merged pull request 893: PLT-5270 Bold, italic and striked links should show link preview when available
- Merged pull request 895: Remove a duplicate max-nested-callbacks key in the .eslintrc.json file
- Merged pull request 989: Migrate delete_post_modal.jsx to be pure and use Redux

Mbedos

- Merged pull request 6286: Small typo fixes in readme.md files

OSU

- Merged pull request 2184: Removing "mouse wheel disabled" checkbox from visual settings in gameplay

Phaser

- Merged pull request 3358: Added rotation, scaling and flipping to TileSpriteCanvasRenderer
- Merged pull request 3357: Fixed object based atlas loading
- Merged pull request 3445: Fix changing alpha in RenderTextureWebGLRenderer
- Merged pull request 3509: Line.PointA&B fix
- Merged pull request 32 in examples repo: Pointerlock example fix

Spark

- Merged commit 6ac4fba for pull request 20880: [SPARK-23769]: [Core] Remove comments that unnecessarily disable Scalastyle check

TypeScript

- Merged pull request 22275: Fix 21617: Give detailed message on `for-of` of iterators without downlevelIteration

Vue.js

- Merged pull request 7739: docs: fix grammar mistake
- Approved fix (not yet merged) in pull request 7938: fix(e2e-todomvc-test): trigger click on .new-todo instead of footer, fix 7937

Xmage

- Merged pull request 4573: Implemented Uphill Battle
- Merged pull request 4584: Typo in classname
- Merged pull request 4617: Fire `PLAY_LAND` event only after replace check
- Merged pull request 4648: Blocker and Critical level bugfixes throughout the project
- Merged pull request 4680: Improved XMage startup time
- Merged pull request 4681: Clickable message of the day
- Merged pull request 4682: Resolving unaccepted changes
- Merged pull request 4683: Fix readme.md card count
- Merged pull request 4707: SOLID violation fix in token classes
- Merged pull request 4709: Fixed test errors caused by Elvish Impersonator