Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering

Master's Thesis

JavaScript Test Runner

Vojtěch Jína

Supervisor: Ing. Jan Šedivý, CSc.

Study Programme: Electrical Engineering and Information Technology

Field of Study: Computer Science and Engineering

June 30, 2013

Aknowledgements

I would like to give special thanks to my mom for persistently forcing me to finish the university; my supervisor Jan Šedivý for his patience and valuable advices; Caitlin for reviewing this text; my colleagues Igor Minár and Miško Hevery for hiring me to the best team I have ever worked with - the AngularJS team, because without AngularJS, Karma would never happened.

Declaration

Abstract

This thesis is about a software for unit testing web applications, called Karma.

The language for web applications is JavaScript, which is a very dynamic language without static typing. There is no compiler that could catch mistakes like misspelling a variable name or calling a non existing method on an object - developers have to actually run the code to catch these issues. Therefore testing is absolutely necessary.

Karma is a test runner, that helps web application developers to be more productive and effective by making automated testing simpler and faster. It has been successfully used on many projects, including companies such as Google and YouTube.

This thesis describes the design and implementation of Karma, and the reasoning behind them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is about testing web applications.

The language for web applications is JavaScript, which is a very dynamic language without static typing. There is no compiler that could catch mistakes like misspelling a variable name or calling non existing method on an object - developers have to actually run the code to catch these issues. Therefore testing is absolutely necessary.

The core part of this thesis is a fully functional implementation of a test runner called Karma. Karma continuously runs all the tests every time any file is changed - in the background, without distracting the developer. This workflow significantly improves the productivity and therefore empowers web developers to rely more on automated testing. As such, Karma has been successfully used on many projects, including companies such as Google and YouTube.

In this thesis, I explain why web developers need such a tool - the problems we are trying to solve. I also describe existing solutions for these problems, including a quick comparison. Later, in the design and implementation chapters, I describe how Karma is designed and implemented, including reasons why I decided on such a design and other options that I had considered.

I believe this thesis will help other people to understand the principles behind Karma.

1.1 Why JavaScript?

The internet has become a very important part of our everyday life. Nearly everything is done online these days, thus web applications have become very important as well. A couple of years ago, HTML and web browsers were only used for displaying static documents; that is gone. JavaScript engines have evolved to be very fast and web applications are everywhere. At this point, JavaScript is still the only language being used in web browsers. There are even projects trying to run a web browser as an operating system [51][49].

JavaScript is a high level language, which makes it very productive. One can create things very quickly, especially in the beginning phase of a project. It is also a very dynamic language, which again makes it flexible and productive. For these reasons many developers continue to get into JavaScript development. Based on the number of projects on GitHub [1], JavaScript is by far the most popular language [10].

However, this dynamic nature makes it more challenging when it comes to building serious products with huge codebases. There is no static typing, which makes type checking and IDEs support hard. Conventions for structure of a project are not settled yet and therefore projects typically end up with very messy codebase. Additionally, there are many inconsistencies between different browsers, especially when it comes to dealing with Document Object Model (DOM).

There are many efforts to fix these issues and improve the web development eco system, such as:

- Browser vendors are improving the browsers by adding new features and making the platform more powerful.
- World Wide Web Consortium (W3C) and TC39 are making the platform more consistent by improving the standards.
- Many libraries (eg. jQuery, Dojo, YUI) are fixing inconsistencies between different browsers by providing higher level APIs.
- IDEs are improving their heuristics for dealing with dynamic languages and bringing refactoring tools for easier coding (eg. WebStorm can understand a JavaScript project very well and offers similar features like the ones for statically typed languages).
- MVC frameworks (eg. AngularJS, Ember.js) are helping with overall application structure and rising the level of abstraction by offering bigger building blocks.
- New languages that compiles to JavaScript (eg. CoffeeScript, LiveScript) are improving the syntax and adding features to the language.
- Other new languages that compiles to JavaScript (eg. TypeScript, Dart) are bringing static types into the language.
- There are even efforts to do a static analysation of JavaScript code and type interferencing (eg. Cloud9) and also efforts to do a runtime type checking.

My goal is to improve the web development eco system by promoting Test Driven Development. This project aims to improve the testing workflows, so that it is easier for developers to test and therefore they do test more.

[1] GitHub is a web-based hosting service for software development projects that use the Git revision control system. As of May 2011, GitHub was the most popular open source code repository site. [50]

1.2 Why Testing?

Test Driven Development (TDD) is still the most effective way how to develop software. At least for me. In the following subsections I describe the reasons for that - what are the benefits of testing.

Note: Originally, TDD meant writing tests before the actual implementation. TDD - as I use the term here - is more about writing testable code and automating the process of running the tests. Whether the developer writes the tests before implementation or afterwards is secondary. However, writing tests first typically produces much better tests.

1.2.1 Proving the Code is Correct

We have always tested our code, it is just a question of how effective it is. When the developer makes a change and then runs the application to see if the change worked as expected, that is manual testing. Even releasing the code to a subset of users is testing. The main reason behind all of these actions is to prove that the code works as expected. TDD is just about making this process fully automated. This is very important, because it means that we can test more often and therefore be more confident that the code is correct.

1.2.2 Avoiding Regressions

This is probably the most known reason behind testing. Regression bug is a bug that had been fixed in the past and then occurred again. Introducing the same bug into the system again can happen very easily, because very often we do not see some hidden relationships. For instance, we change obviously unrelated piece of code and therefore we do not check some old problem, because we do not expect that problem to occur again. Once we have an automated test for this bug, it will not happen again (at least the probability is very low), because we can easily run all the tests, instead of manually trying only the parts that are obviously related to the change we made. This gives us more confidence when making any changes and therefore it makes things like refactoring easier.

1.2.3 Quick Feedback

In order to be creative and productive, a quick feedback is required [46]. Whenever you make any changes, you want to immediately see the result of that change. For example, when a designer is drawing on a paper - he draws a line and immediately sees what the result is enabling him to immediately decide whether he likes it or not. When writing a software, especially using compiled languages, this is not the case. The developer makes a change and then he needs to wait for compiling the code and then running it. That is very slow feedback, slowing the whole development process and killing creativity. JavaScript is an interpreted language and reloading the browser is usually pretty fast, at least compared to compiling C++ code, but it could be faster. Running unit tests can be way faster than switching context from the text editor to the browser and waiting for the page to reload. Executing a unit test can happen in a few milliseconds, without the developer leaving his text editor at all, creating instant feedback from testing.

1.2.4 Safer Refactoring

Refactoring is a very important part of the development process. You might be an amazing architecture designer, but your architecture will not work forever. Usually, the architecture is great at the beginning, but then changes to a project requirements inevitably cause the original architecture to no longer fit well. I think this is absolutely natural and common. The only way to cope with these changes, is to keep refactoring and adapting our architecture. Unfortunately, refactoring can be very dangerous. It is very easy to break existing code by refactoring and therefore people usually do not do it. Testing gives you the confidence. Tests prove whether the code still works, even after a major refactoring. This confidence is very important, because it empowers people to do refactoring and keep the architecture fresh.

1.2.5 Documentation

A well written test tells a story - a story about how to use the code under test, therefore tests can serve as a documentation. There is one big advantage over the classical documentation: this documentation will not get stale. As long as the tests are passing, the documentation is up to date.[2]

1.2.6 Better Design

Testing is not about writing tests. It is about writing a testable code.

In order to test a single unit, we need to be able to instantiate that unit standalone - without the rest of the system. That is important, because instantiating the entire system would be too expensive and would make the tests slow. It is also necessary to instantiate the same unit multiple times, using different configurations and different collaborators, so that we can test that the unit behaves correctly in all the different situations. Without the ability to instantiate each unit standalone, testing correct behavior in all the different situations would be nearly impossible. This leads to **a loosely coupled code[3] with explicit dependencies**.

Typically, some parts of the system are more expensive than others. For instance accessing network resources is more expensive than pure computation. Expensive means it is more work to set it up and also slower to execute. Therefore it is important to separate the cheap parts from the expensive ones, so that it is possible to test the cheap parts without the expensive ones. A typical example from web development is separating pure JavaScript logic from DOM operations. This leads to **a better separation of concerns**, such as separating view from logic, and also following **the single responsibility principle**.

It turns out that these attributes of testable code are also attributes of reusable code, which is easier to maintain and also easier to understand. In general, testing forces developers into a better design.

[2]A great example of such a documentation is Jasmine [38].

[3]Loosely coupled system is one in which each of its components has very little or no knowledge of the definitions of other components.

Chapter 2

Description of the Problem

In this chapter, I describe that web developers do not have sufficient tools to do testing effectively and why having these tools is important for producing a high quality software. In the second part of this chapter, I describe concrete goals for this project - how I want to improve the current experience and promote testing in web development by creating a new test runner.

Web developers do not have sufficient tools for automated testing and therefore they do not rely on testing. Very often they do not run automated tests at all.

In other languages and environments like Java, Python or C#, the testing tools and their integration with common IDEs is pretty decent - the developer can easily run tests directly from the IDE and almost instantly see the results. In the case of statically typed languages like Java or C++ there is also the compiler that can discover a lot of problems before even executing the code.

We focus on web developers and therefore the language is JavaScript. JavaScript is a very dynamic language without static typing. There is no compiler that could catch mistakes like misspelling a variable name or calling non-existing method or reading non-existing property of an object. The developer has to actually run the code to catch any of these issues. Therefore testing is absolutely necessary - it is just a question of how much can be automated.

Web applications run in a web browser. This means that in order for the developer to test if a code change did not break the system, he needs to open the browser, load the app and visually check if it works. **This is the core issue - the workflow**. The missing compiler is not that big of an issue, because the JavaScript virtual machines in web browsers have typically very quick bootstrap and so running the code can be faster than compiling C++ or Java code. The core issue is the workflow - the developer has to switch context from the editor to the browser, reload the page and visually check the app, probably even open the web inspector and interact with the app to get it into a specific state.

On the top of that, there are multiple browsers with different issues - especially when it comes to dealing with DOM. Therefore testing the app in one browser does not necessary mean it works in the other browsers as well. This whole experience can get really painful.

It is possible to execute JavaScript without a browser, using environments like Node.js [36] or Rhino [54]. That can improve the workflow quit a bit. However, it has two problems: First, the developer has no access to browser specific stuff such as DOM and its APIs, which limits to only testing pure JavaScript logic. Second, different browsers have different issues. In order to catch these issues, we need to execute the tests on all targeted browsers.

2.1 Goals

The goal of this thesis is to create a tool - a test runner, that helps web application developers to be more productive and effective by making automated testing simpler and faster.

In fact, I have a much higher ambition and this thesis is only a part of it - I want to promote Test Driven Development (TDD) as "the" way to develop web applications, because I believe it is the most effective way to develop high quality software.

There are two essential prerequisites that the developer needs in order to successfully apply TDD:

- testable code
- testing environment

Writing testable code, is where AngularJS [1] and the philosophy behind it come in. AngularJS guides developers to write loosely coupled code that is easy to test. Furthermore, every tech talk and demo that we as the AngularJS team give, follow the testing philosophy.

Karma applies to the second prerequisite - its goal is to bring a suitable testing environment to any web developer. The overall setup and configuration has to be straightforward, so that it is possible to start writing tests in a few seconds and instantaneously see test results on every code change.

In the next subsections I describe use cases and concrete features that Karma needs to support. I also explain why these features are important.

2.1.1 Testing on Real Browsers

There are many cross-browser issues and one of the goals of testing is to catch these issues. A cross-browser issue means that some browsers implement an API slightly differently or does not implement some features at all. Different browsers running on different Operating Systems (OS) can also have different bugs. That is why Karma needs to be able to execute the tests on any browser and even on multiple devices such as a phone or a tablet.

[1]AngularJS [11] is a JavaScript framework that I have been working on over the past two years.

2.1.2 Remote Control

One of the core issues of web development is the workflow. The developer has to switch context all the time; i.e. make a change in the editor, go back to the browser, refresh the page, interact with it, see if it works. Then, go back to the editor, make some more changes and repeat. This is a distraction which we want to eliminate.

The basic workflow of running the tests on every save should work automatically without any additional interaction. The developer should just save a file and instantly see test results, without leaving the text editor.

For other tasks such as triggering the test run manually, Karma needs to be fully controlled from a command line. That will allow easy control from any IDE as well.

2.1.3 Speed

Waiting for test results can be very distracting, as it can easily take more than a few minutes. Therefore developers tend to run the tests only several times a day, which makes it very difficult to debug possible test failures. The reason for that is simple - the more code the developer wrote since the last test run, the more space to look for the error.

I want to minimize this space by allowing developers to run the tests on every file save. If the developer runs the tests every time a file is saved, it is easy to spot the mistake, because the amount of code written since the last run is small.

In order to make this useful, the test execution has to be fast. The developer can not wait more than a few seconds. That is why Karma has to be fast.

In addition, Karma needs to provide a simple mechanism of focusing on a subset of tests. For instance, if the developer is working on a specific feature, there needs to be a way to run only tests that are directly related to that feature. Huge projects can have thousands of tests and running all of them will take more than couple of seconds. Hence running only a subset of tests will allow frequent test execution even on huge projects.

The core idea is to get a continuous feedback from tests, with minimal distraction to the developer.

2.1.4 Integration with IDEs and text editors

Karma needs to be agnostic of any text editor or IDE. It needs to provide basic functionalities such as watching source files and running the tests on every change or running only a subset of tests, without any support from the text editor or IDE.

2.1.5 Integration with CI Servers

Continuous Integration (CI) was originally a practice of merging all developers workspaces with a shared code repository several times a day to prevent integration problems. These days, the main reason for having a CI server is that building the whole project and testing on all the devices usually takes a long time and therefore developers do not spend time doing it on their machines. Very often they also do not have all of these devices available. As a result, developers typically run only some lightweight test suite and rely on the CI server to run the whole suite while they are working on something else. Typically, the CI server would run this on every push to the repository or in some cases even before pushing.

Debugging failures on a CI server is very difficult as it is typically a remote machine with restricted access. Therefore Karma needs to integrate with CI servers so that it is possible to use the same configuration both locally during development and on the CI server.

In order to integrate well with most common CI servers, Karma needs to be able to produce test results in XML and provide a special mode in which it only executes all the tests once and immediately exits with a proper status code, representing either success or failure.

2.1.6 Extensibility

Karma should allow writing plugins, that can extend the core functionality and enable easy integration with any framework or library. For example file preprocessing, so that other languages such as CoffeeScript [6] or LiveScript [2] can be used or plugins for dependency management systems such as RequireJS [8] or Google Closure [12].

2.1.7 Debugging

In general, debugging means removing bugs from a system. Most of the environments allow running code in a special mode, where the developer have better access into internals of the program. For instance, breakpoints can be inserted into the code to pause the execution. This allows the developer to inspect the call stack, variables and step the program execution line by line. This is crucial when trying to understand behavior of the program - it allows the developer to follow exact steps and inspect the states that the program actually goes through.

Typically the debugger is integrated into an IDE, which makes it very convenient for the developer - the developer can stay in the IDE, put a breakpoint on a specific line, run the debugger and inspect the program and its state, without leaving the IDE.

A unit test typically instantiates an instance of an object, performs some action on it; i.e. calling its methods, and then asserts the state or whether some other action happened. If such a test fails, it probably means that the object under test did not work correctly, but it is not always that simple to understand why. The developer does not see internals of the program, he only sees that the eventual output was something different than what was expected. That is when debugging becomes very handy and is the reason why Karma needs to support debugging.

Most modern web browsers already have a built-in debugger. Karma needs to enable using the browser debugger - both directly inside the browser or through IDEs (eg. WebStorm can do that).

Chapter 3

Existing Solutions

In this chapter, I describe existing solutions for testing web applications and explain what are their advantages and disadvantages, along with their sweet spots. At the end I compare all of these solutions.

Testing can be done on multiple levels - from very focused low level tests that exercise a single unit without the rest of the system, to high level tests that exercise the whole system from an end-user point of view. Low level tests typically do not require much set up making them fast to execute. It is not possible however, to cover all requirements using only low level tests - for instance testing whether or not all components communicate correctly. In addition, integration with a third party service is almost impossible without high level tests. High level tests require more time to execute. In order to make testing efficient, we need to test on multiple levels.

The following sections describe existing solutions for testing web applications. Most of these solutions are suitable for either low level testing, such as Mocha or JsTestDriver, or high level testing, such as Selenium or WebDriver. Therefore these solutions are not necessarily competing - they are rather complementing each other.

3.1 Selenium

Selenium [3] is an entire suite of tools. It is one of the oldest tools making it very mature. It is suitable for high level testing, where testing the whole application from an end-user point of view.

The core idea is called "proxy injection". Selenium opens a browser with proxy set to a local URL, where the Selenium server is listening. Whenever the browser opens any URL, it always sends the request through that proxy (Selenium server), which fetches the originally requested page and injects a piece of JavaScript code into it. This injected JavaScript code manages the communication with the server.

The developer uses the Selenium client library which provides APIs such as "navigate to URL" or "click a button" to write the actual tests. This client library executes these tests and sends commands through HTTP to the Selenium server which has a connection with the browser and forwards these commands to the browser. The injected JavaScript code interprets these commands in the browser.

There are multiple implementations of the client library for most of the common languages such as Java, Python, Ruby. As a result of that, the developer can use any of these supported languages to write the tests.

3.2 WebDriver / Selenium 2

WebDriver [4] is a HTTP based protocol for communication with a web browser. Selenium 2 is a new generation of original Selenium and it is an implementation of WebDriver protocol [47].

When available, it uses native driver which brings a better control over the browser, such as triggering native DOM events. For browsers that do not support native driver, such as Safari, it uses the old "proxy injection" technique from Selenium 1.

3.3 Mocha

Mocha [17] is a testing framework and a test runner, that runs JavaScript code in Node.js (Node) environment, which is the main use case for it - low level testing Node projects. I think it is the best test runner available for this purpose.

Mocha can be fully controlled from the command line interface which is important for a good workflow. As such, Mocha can easily be integrated with any IDE. Figure 3.1 shows an example output of Mocha runner.

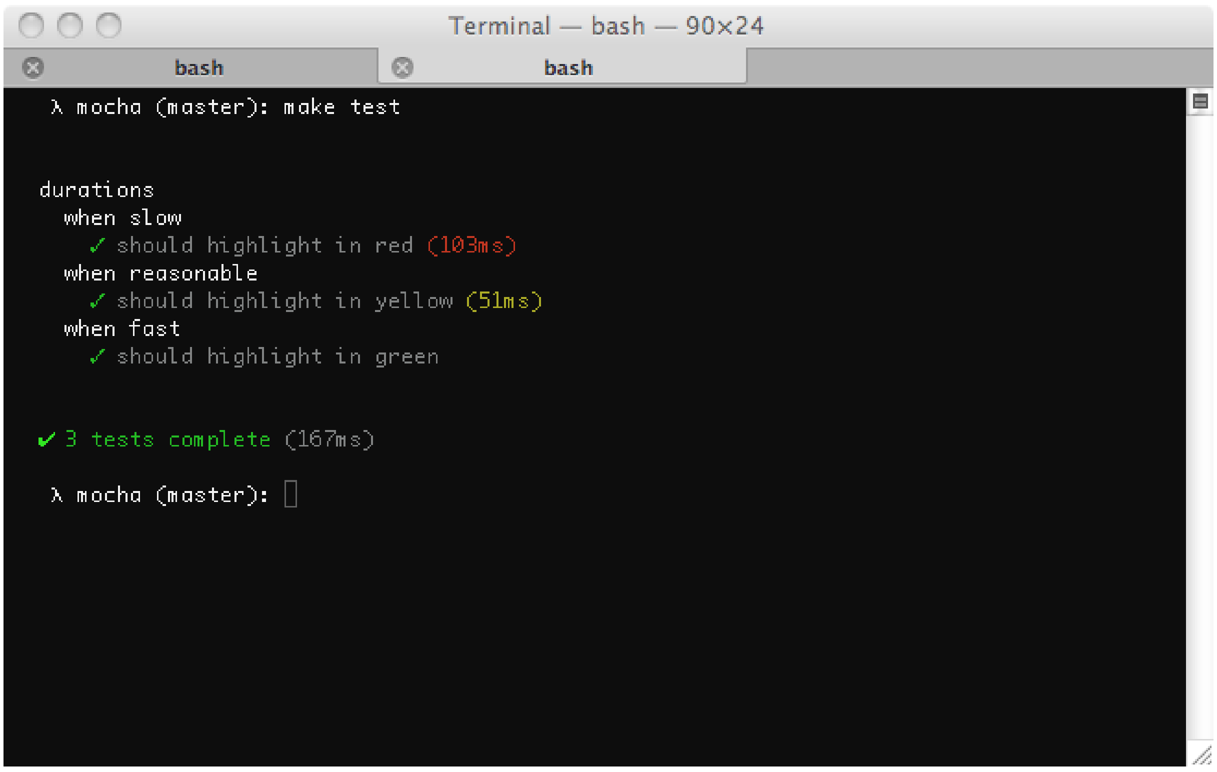
It is possible to test client side JavaScript with Mocha too, but with a limitation to pure JavaScript - there are no browser APIs such as DOM. This means that no cross-browser issues can be tested in this way. As discussed in the previous chapter, this is crucial and makes testing web applications with Mocha is insufficient.

Note: Mocha can run in a browser as well, in the same way other HTML runners do. That means it suffers from the same problems as well - poor workflow. The developer has to open up a browser and reload the runner anytime he wants to run the tests.

3.4 JsTestDriver

The main use case for JsTestDriver [15] (JsTD) is low level testing. The tests are executed directly in the browser and therefore there is direct access to the JavaScript code, DOM and all the browser APIs.

The developer starts a JsTD server and points a browser to the URL, where the server is listening. JsTD then captures the browser - establishing a connection with the browser. Anytime the developer wants to run the tests, he runs JsTD runner, which notifies the server. The server sends a command to the browser, which reloads the JavaScript files (served by



```
 λ mocha (master): make test


durations
  when slow
    ✓ should highlight in red (103ms)
  when reasonable
    ✓ should highlight in yellow (51ms)
  when fast
    ✓ should highlight in green


✔ 3 tests complete (167ms)

 λ mocha (master): 
```

Figure 3.1: Mocha runner output in the command line

the JsTD server), executes all the tests in the browser and reports the results back to the JsTD server.

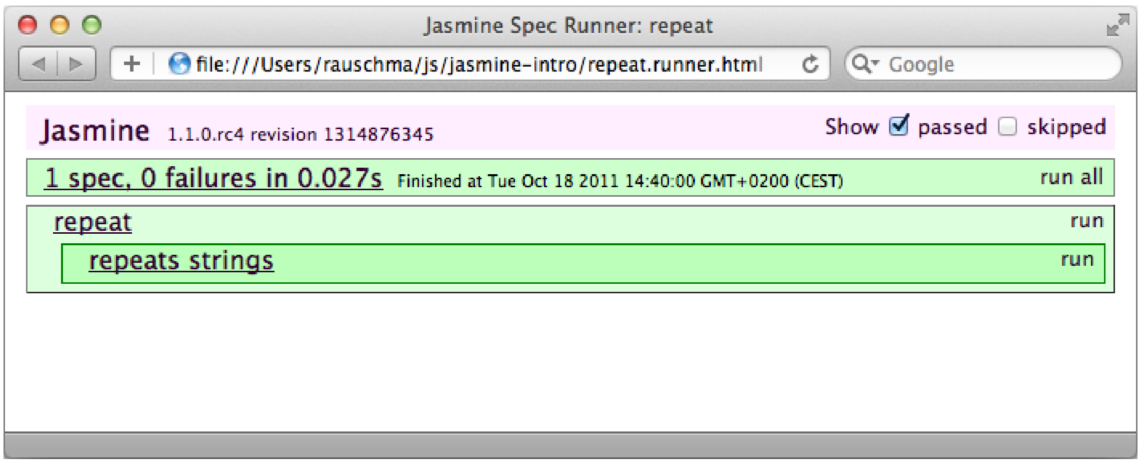
3.5 HTML Runners

Most of the testing frameworks such as Jasmine [38] or QUnit [37] come with a basic HTML runner. An HTML runner runs in a web browser and reports the test results by rendering some HTML that presents the results. Figure 3.2 shows an example output of a test run using Jasmine HTML runner.

There is no other component - the developer does not have to start any server, but he has to switch to the browser and refresh the page every time tests need to be executed. In addition, the developer needs to maintain an HTML file that lists all of the JavaScript source files.

3.6 Comparison

In this section I summarize all the previously mentioned solutions and compare them. These tools were not designed for the same job and therefore the comparison is more less


Jasmine Spec Runner: repeat
file:///Users/rauschma/js/jasmine-intro/repeat.runner.html
Google

Jasmine 1.1.0.rc4 revision 1314876345 Show ✓ passed ☐ skipped

1 spec, 0 failures in 0.027s Finished at Tue Oct 18 2011 14:40:00 GMT+0200 (CEST) run all

repeat run

repeats strings run

Figure 3.2: An example of Jasmine HTML runner output

describing what the sweet spot for each of those tools is. Table 3.1 shows a side-by-side comparison of all mentioned solutions and their individual features.

Selenium and WebDriver do not provide direct access to the JavaScript code of the app, because the tests are executed on the server and the browser only interprets individual commands. On the other hand, WebDriver has a big advantage of triggering native DOM events. That is very difficult to simulate in JavaScript, which makes WebDriver / Selenium 2 suitable for high level testing, where we test the app from an end-user point of view. As such, WebDriver is becoming the standard way of high level testing.

HTML runners are just an additional feature to testing frameworks, making it is possible to run the tests without the installation of another tool for this task. The main issue from the developer productivity point of view is the fact that there is no remote control - the developer has to switch context to the browser and refresh the page in order to run the tests. Furthermore, there is no integration with IDEs or CI servers.

Mocha provides a really good workflow and from that point of view it is very close to what we want. However, it only executes in the Node environment (running Mocha in a browser falls into the HTML runners category) and therefore there is no way to test against browser specific APIs such as DOM. Nor is there a way to test cross-browser inconsistencies. As discussed in the previous chapter, this is a crucial requirement for testing web applications.

JsTestDriver is the closest solution to what we are looking for. It is suitable for low level testing. It runs the tests in the browser and therefore give us a direct access to the JavaScript code, DOM and all browser APIs. It also has a remote control to trigger a test run from the command line.

However, JsTestDriver has two major problems. First, its design and implementation has several flaws which makes it slow and unreliable. Second, it is missing features such as watching source files or preprocessing source files to make the final workflow seamless.

Karma improves upon JsTestDriver by bringing new features such as watching files or file preprocessing. Watching the files is very important, because it enables integration with any text editor. The developer does not have to trigger a test run manually - simply saving a file

	suitable for	direct access to JavaScript	DOM API	remote control	file watching	file preprocessing	tests written in
Karma	unit	yes	yes	yes	yes	yes	any[1]
JsTestDriver	unit	yes	yes	yes	no	no	JS
Selenium	e2e	no	yes	yes	no	no	any
WebDriver	e2e	no	yes	yes	no	no	any
Html Runners	unit	yes	yes	no	no	no	JS
Mocha	Node	yes	no	yes	yes	yes	JS

Table 3.1: Comparison of existing solutions and their features

triggers a test run automatically. File preprocessing makes it easier to use other languages which compiles to JavaScript, such as CoffeeScript [6], TypeScript [40] or Dart [13]. File preprocessing can also be used for pseudo compiling HTML into JavaScript strings, which significantly simplifies testing with HTML templates.[2]

As discussed above, the main issue of JsTestDriver is its unreliability, causing the developer to need to pay a lot of attention to the tool itself. Karma solves this problem with a different design (described in 4 and 5), which makes it reliable and consistent. Errors in code under the tests do not affect stability of the runner.

Another important feature is speed - Karma is designed to be fast.

All these features share a single goal - **a seamless workflow**, which provides fast and reliable results, without distracting the developer. That is the core new feature that Karma brings into the web development.

[2]There is an example project that shows this feature with AngularJS templates [32].

Chapter 4

Design / Analysis

In this chapter, I describe the overall design of the proposed architecture as well as reasons why I decided on this design. In the end of this chapter I also discuss some of the other design options that I considered, including their pros and cons.

The main goals that drove my design and implementation decisions were:

- speed,
- reliability,
- testing on real browsers and
- seamless workflow.

The overall architecture model of the system is client-server [48] with a bi-directional communication channel between the server and the client. Typically, the server runs on the developer's local machine. Most of the time, the client runs on the same physical machine, but it can be anywhere else, as long as it can reach the server through HTTP.

A single instance of the server has only notion of a single project. If the developer wants to work on multiple project simultaneously, he needs to start multiple server instances.

Figure 4.1 shows a high level diagram of the overall architecture, including the individual components of the server and the client. In the following two sections, I describe the server and the client design in more details.

4.1 Server

The server component is the master part of the system - it keeps all the state (eg. information about captured clients, currently running tests or files on the file system) and operates based on the knowledge of that state. It has many responsibilities such as:

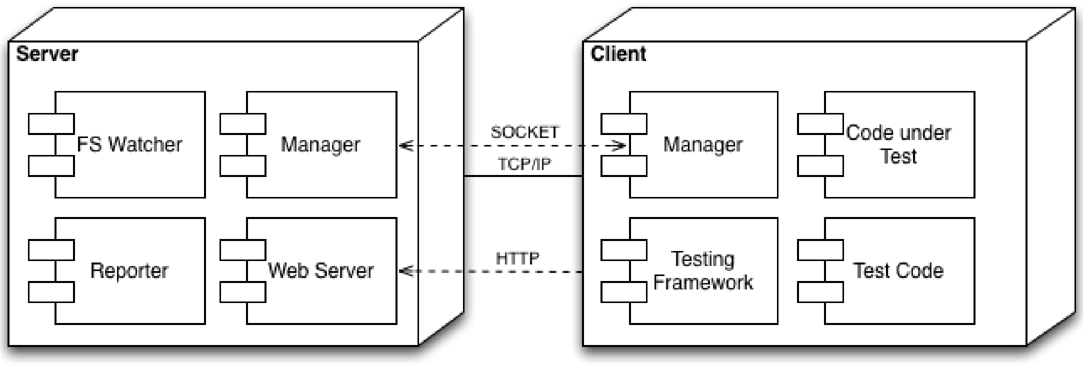


Figure 4.1: The high level architecture

- watching files,
- communication with captured clients,
- reporting test results to the developer and
- serving all the client code.

Note: There can be many clients connected to the server. A client is a web browser, typically running on the same physical computer as the server, but it can be a web browser running on a mobile phone, tablet or TV as well. The server is the master and clients are slaves, therefore from here, I will call them "captured clients".

The server component runs on the developer's machine so that it has fast access to the file system where all the sources and tests of the project are. The server knows what files the project under test consist of based on a project configuration, which is described in more details in B.2.

The server consists of four major components:

4.1.1 Manager

The manager is responsible for bi-directional communication with captured clients. For instance, broadcasting the signal to start test run or handling messages from clients such as collecting the test results.

The manager maintains an internal model of all captured clients (eg. name and version of the browser, whether it is currently idle or executing tests) and an internal model of a test run (eg. how many tests have been already executed and how many of them failed).

As the communication gateway it is also responsible for notifying other components about changes in these models. For instance when there is a new test result from a client, the reporter needs to be notified.

4.1.2 Web Server

The web server is responsible for serving all the static data required by the client. The static data is source code for the client manager, the source code of the testing framework as well as the test code and the actual code under test.

4.1.3 Reporter

The reporter is responsible for presenting the test results to the developer. It generates the output based on the changes in the test results model. Typically the output is writing test results on the screen or into a file for the purpose of a CI server.

4.1.4 File System Watcher

The watcher is responsible for watching the file system (FS) - it maintains an internal model of the project under the test (all its files, where they are located and timestamps of their last modifications).

This FS model is consumed by the web server in order to serve correct files to the clients. It also allow us to minimize FS access and network traffic - we only reload the files that changed. This is described in more details in 5.4.

4.2 Client

The client is the place where all the tests are actually executed. It is typically a web browser on any device such as a computer, a phone or a tablet. It usually runs on the same physical machine as the server, but can run anywhere as long as it has access to the server through HTTP.

There can be multiple client instances communicating with the same server.

4.2.1 Manager

The manager is responsible for bi-directional communication with the server. It handles all the messages from the server (such as triggering the test run) and communicates them to other client components (usually the testing framework).

4.2.2 Testing Framework

The testing framework is not part of this project. Karma is flexible enough to allow using any third party testing framework out there.

4.2.3 Tests and Code under Test

This is all user's code that is run by the testing framework. It is fetched from the web server and executed through the testing framework.

4.3 Communication Protocol

This section describes the bi-directional communication between a client and the server.

4.3.1 Client to Server Messages

- At the beginning, when a client is being captured, it needs to send its identification number as well as other information such as the browser name and version.
- When a single test case finishes, the client sends a result message. This message contains results of the test such as if it succeeded or failed.
- When all tests are finished the client sends a message.
- When any error happens, either during bootstrap (eg. syntax error) or during the actual test run, the error is sent to the server.

4.3.2 Server to Client Messages

- When the server decides to start a test run, it sends a signal to all captured clients.

4.4 Other Considered Design Options

In this section I describe some of the other design options that I considered and why I did not decide to choose these options.

4.4.1 Single Component vs. Client-Server

As described in the previous sections, the overall architecture model of the system consists of two components - client and server.

Another solution could be a single component, that executes the tests in an environment such as Rhino or Node.js. That would be similar to most of the other testing tools that are usually part of IDEs for languages like Java or C#. In such a scenario, the IDE knows where the VM is located and whenever the developer wants to run the tests, they can be run on this VM directly from the IDE. Some IDEs even provide plugins for running on every file save.

Web development is quite different here, as there are many inconsistencies between different browsers running on different devices and this kind of environment was our primary target. Separating the client (where the actual test execution happens) and communicating through HTTP makes it possible to test on any web browser supporting HTTP - and that is every web browser running on computer, phone or tablet. The client-server model also allows execution on multiple devices in parallel and makes it easier to add new clients without changing the server, as the client and server are loosely coupled.

4.4.2 Multiple Projects Support

A single instance of the server has only notion of a single project. If the developer wants to work on multiple project simultaneously, he needs to start multiple server instances.

At first, I thought about supporting multiple projects per single instance of the server. The developer would run only single instance of Karma and use it to test multiple projects.

Advantages of supporting multiple projects per single server instance:

- The developer does not have to start new instance per project.
- Less memory consumption (only single VM running)[1].

Disadvantages:

- Significantly increases implementation complexity.
- All the projects have to depend on the same version of Karma.

Simplifying the design was the main reason for my decision to only support a single project per server instance scenario.

[1]However, this is not a big deal as the whole Karma usually takes about 60MB of RAM.

Chapter 5

Implementation

In this chapter, I describe the actual implementation of the design proposed in previous chapter as well as reasons why I decided for particular implementation. Instead of describing all the details, I describe just the high level overview and then I spend more time with detailed description of interesting parts of the system.

Karma is implemented in JavaScript. The server part runs on Node.js (Node) [36] and the client part runs in a web browser. There were many reasons why I decided for Node, such as:

- It runs on all major platforms.
- Entire codebase is in one language (the client code runs in a web browser where JavaScript is the only choice).
- It is a very productive environment.
- I wanted to explore asynchronous programming in more depth.

In the following sections I describe implementation of the two major parts of the system - the server and the client - and how they communicate.

5.1 Server

This section describes the implementation of the core components of the server. The functionality and responsibilities of the server and its components is described in the design chapter 4.1. Figure 5.1 shows all the server components and how they communicate. The solid lines represent direct method invocation, the dashed lines represent communication through events.

In the following subsections I describe implementation of these components in more detail.

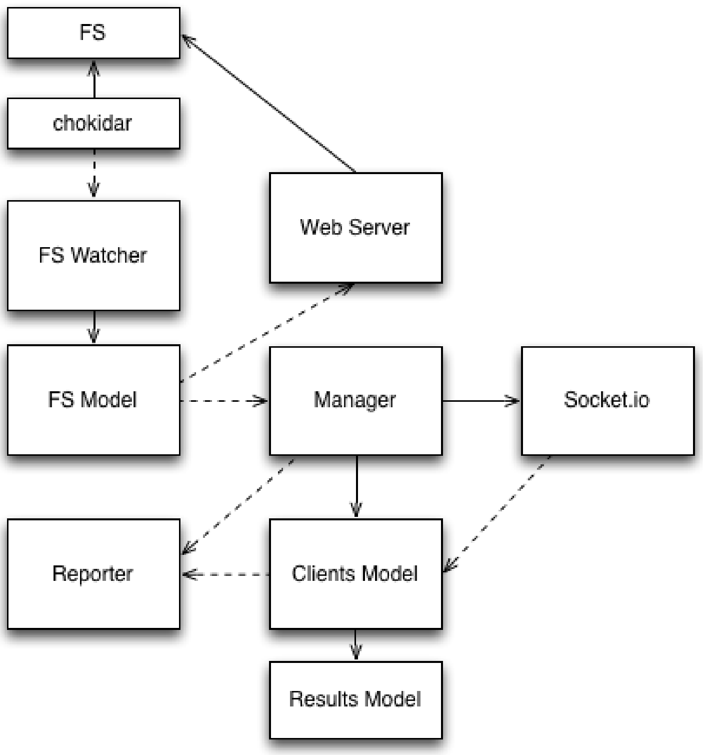


Figure 5.1: Server components

5.1.1 File System Watcher

The watcher is implemented using the chokidar [41] library, which is a wrapper around Node's native API (`fs.watch` and `fs.watchFile`). During the bootstrap, the watcher configures chokidar to watch particular files and directories on the FS. Everytime any of these files or directories changes, chokidar emits an event. The watcher is listening on these events and maintains an internal FS model. The watcher maintains the model through its API (described in figure 5.2) by direct method invocation.

Chokidar library abstracts away inconsistencies between different platforms. It also provides higher level APIs that are not supported by Node's native APIs. For instance observing changes in entire directories.

Internally, Node uses system events such as inotify[1] or kqueue[2], if given OS supports it and falls back to stat polling if the system events are not supported.

5.1.2 File System Model

The main purpose of the internal file system (FS) model is to minimize access to the real FS as well as network traffic. See 5.4 section for more information on caching.

The FS model (internally represented as a single instance of `FileList` object) contains meta data about the project under test - about its source files. These files are specified in the config file as a list of glob patterns (eg. `js/src/*.js`, `test/**/*Spec.js`). `FileList` internally maintains a list of buckets - a bucket represents a single pattern that can match multiple files. A bucket contains a list of `File` objects, each representing a real file matched by this pattern. This concept of buckets is important because of the order of the files has to be predictable. Therefore `FileList` keeps the original order of patterns (as defined by the developer in the config file) and sorts files inside a bucket (matched by a single pattern) alphabetically.

Whenever there is any change to the FS model, such as adding a new file, removing a file or changing a file, `FileList` communicates these changes to the rest of the system. It emits `file_list_modified` event and pass a single argument along with the event - a promise containing all the files in correct order. There are two observers listening to this event:

- **web server** (described in 5.1.3) uses the promise to determine which files to serve, where they are located and for generating the `context.html`,
- **manager** triggers a test run.

This promise does not contain buckets anymore. It is a list of files in the correct order. For more details on how this is implemented, see 5.5.

Figure 5.2 shows the public API of `FileList`. This API is used by the watcher.

[1] Inotify is a Linux kernel subsystem that acts to extend filesystems to notice changes to the filesystem, and report those changes to applications. [52]

[2] Kqueue is a scalable event notification interface. [53]

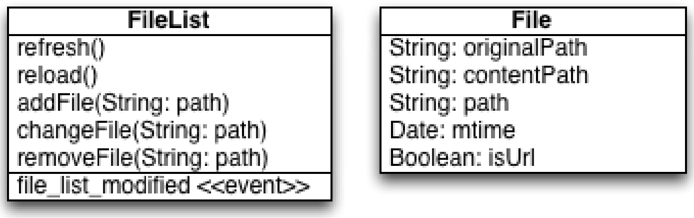


Figure 5.2: File System Model

5.1.3 Web Server

The web server is implemented as a list of handlers. When a request arrives, the first handler is called. Each handler can either handle the request and generate the response or call the next handler.

Code 5.1 shows an example of such a handler. The first argument is `http.IncomingMessage` object that contains information about the request, such as which url has been requested. The second argument is `http.ServerResponse` object, essentially a writable stream. If the handler knows how to handle that request, it can write the response into that stream. The last argument is a function - a next handler. If this handler does not know how to handle the request, it passes the request to the next one by calling this function.

```
var exampleHandler = function(request, response, next) {
  if (knowHowToHandle(request)) {
    response.write('something');
    response.end();
  } else {
    next();
  }
};
```

Code 5.1: Example of a request handler

Typical solution would be to have a global router that decides who will handle the request and then call that handler. I decided to implement the web server as a list of handlers, because **it moves the responsibility of the decision to the handler** - the handler itself decides whether it knows how to handle a particular request. It is also more functional (a handler gets function next rather than returning a value) and therefore **it can be completely asynchronous** - both the decision whether it handles the request and the actual request handling - producing the response - can be asynchronous. This design also makes it easy to extend the functionality of the web server by a plugin that can add its handler.

There are four handlers:

- karma client files (all the code required for the communication with the server)
- testing framework and adapter
- source and test files
- proxy

The web server also consumes the FS model through `file_list_modified` event. Based on the FS model, it knows which files to serve, which files to include in the `context.html` and where on the real FS are all the files located.

The web server also contains a proxy, which is implemented using http-proxy [42] library.

5.1.4 Reporter

The reporter observes any change in the Results model and reports them to the developer. Code 5.2 shows an example of a simple reporter that displays OK or FAIL on the terminal output. It is possible to write a custom reporter and plug it into the system as a plugin.

```
var SimpleReporter = function() {
  this.onSpecComplete = function(browser, result) {
    console.log(browser.name + (result.success ? 'OK' : 'FAIL');
  };

  this.onRunComplete = function() {
    console.log('All browsers are done.');
  };
};
```

Code 5.2: Example of a simple report

5.2 Client

The client part of the system is where all the tests are actually executed. The functionality and responsibilities of the client and all its components is described in the design chapter 4.2.

The client is more or less a web app, that communicates with the server through socket.io [43] and executes the tests in an iframe. Figure 5.3 shows the overall client structure. In the following subsections, I describe the implementation of individual components. The communication between client and server is described in section 5.3.

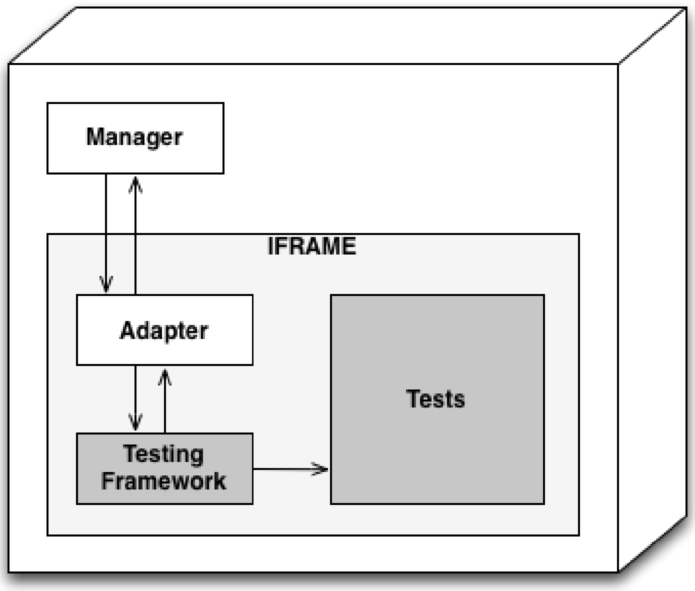
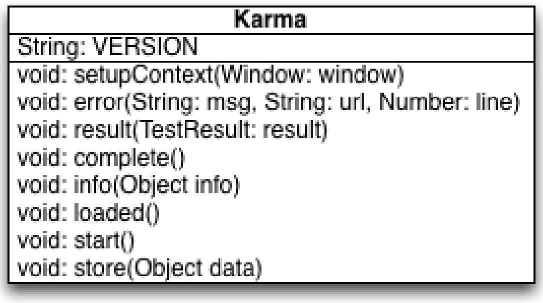


Figure 5.3: Overall structure of the client

5.2.1 Manager

The client manager is implemented as a thin layer on the top of socket.io library. It provides the client API for particular testing framework adapter to communicate with the server. Figure 5.4 shows the client API. Code 5.3 shows an example of a testing framework adapter, which uses the client API.



```
Karma
String: VERSION
void: setupContext(Window: window)
void: error(String: msg, String: url, Number: line)
void: result(TestResult: result)
void: complete()
void: info(Object info)
void: loaded()
void: start()
void: store(Object data)
```

Figure 5.4: Client API

The client manager runs in the main HTML frame, but the API is exposed into the iframe where the actual test execution happens. Lifetime of the client manager is multiple test suite runs - it lasts until the client is shut down or manually restarted by the developer. Therefore it is crucial to take a good care of memory management.

5.2.2 Iframe

The client contains a single iframe where all the test execution is done. Triggering a test run means reloading this iframe. The source of the iframe is `context.html`, which is an HTML file containing a bunch of `<script>` tags with all the included files. All the tests and source files, the testing framework and its adapter are loaded inside this iframe.

Complete reload of the iframe is important as it means a fresh context for every test suite run. We could keep everything in the main frame and only reload the files that changed. That would be faster, but not reliable. Tests can access global window and change its state and therefore a test suite run can cause failure of a test in a subsequent test suite run[3]. That was the main reason why I decided to always reload the iframe and rather rely on heavy caching mechanism described in the section 5.4.

Ideally, we could use a fresh iframe per test. That would allow people to write nasty code and tests with mutating global state without any problems. However, it would make the tests much slower and therefore I think it is better to evangelise proper way of writing code using Dependency Injection pattern and avoiding mutable global state.

5.2.3 Adapter

A testing framework provides Domain Specific Language (DSL) for defining tests. There are many testing frameworks, supporting all kinds of different styles. The choice of style usually depends on a personal preferences of each developer. Therefore I decided to not make Karma dependent on a specific style, and rather let the developer to choose his preferred testing framework. Consequently, Karma integrates with existing solutions through adapters.

Each testing framework has a different API for executing tests and reporting the results. An adapter is basically a wrapper around the testing framework that translates the communication between the testing framework and Karma client manager API.

Code 5.3 shows an example of such an adapter. An adapter has to implement method `__karma__.start`. Karma calls this method when starting test execution. At that point, the adapter instantiates Jasmine HTML runner and passes it a custom reporter (`JasmineToKarmaReporter`). This reporter gets called by Jasmine - for instance `reportSpecResults` is called after each test finishes, `reportRunnerResults` is called after the whole test suite finishes. The `JasmineToKarmaReporter` does translate each method call into Karma Client API described in figure 5.4.

5.3 Communication Between Server and Client

The communication between client and server is implemented on the top of socket.io [43] library. The socket.io library provides an event based communication channel. Code 5.4 shows a simple example of such a communication - server manager emits events that client manager is listening on and vice versa.

The socket.io library implements multiple protocols such as WebSocket, XHR polling, HTML polling, JSONP polling and uses the best one available on given web browser.

[3]This is a typical issue with JsTestDriver, which does not reload the frame.

```
var JasmineToKarmaReporter(clientManager) {
  this.reportSpecResults = function(spec) {
    var result = {
      id: spec.id,
      description: spec.description,
      success: spec.results_.failedCount === 0,
      skipped: spec.results_.skipped
    };

    // call the client manager API
    clientManager.result(result);
  };

  this.reportRunnerResults = function(runner) {
    clientManager.complete();
  };
};

window.__karma__.start = function() {
  var jasmineEnv = window.jasmine.getEnv();
  var reporter = new JasmineToKarma(window.__karma__);

  jasmineEnv.addReporter(reporter);
  jasmineEnv.execute();
};
```

Code 5.3: Example of an adapter between Jasmine testing framework and Karma

Messages are encoded as JSON. Code 5.5 shows the message from code example 5.4, encoded into JSON. That is how the message is actually transferred over the network.

```
{"name": "message", "args": [{"hello": "world"}]}
```

Code 5.5: Message from previous example (encoded as JSON)

5.3.1 Client to Server Messages

The following table 5.1 shows a list of implemented events between client and server - these messages are sent from client to server. Calling the client API methods described in figure 5.4 emits these events. Socket.io layer is responsible for encoding the data into JSON and sending them to the server. On the server, there is the server manager listening on these events and dispatching them to interested server components.

```
// server manager
var io = require('socket.io').listen(80);

io.sockets.on('connection', function (socket) {
  console.log('new client connected');
  socket.emit('message', {hello: 'world'});
});

// client manager
var socket = io.connect('http://localhost');

socket.on('message', function (data) {
  console.log('a message from the server', data);
});
```

Code 5.4: Example of basic communication using socket.io library

event	*description*
register	the client is sending its id, browser name and version
result	a single test has finished
complete	the client completed execution of all the tests
error	an error happened in the client
info	other data (eg. number of tests or debugging messages)

Table 5.1: Events emitted by client

5.3.2 Server to Client Messages

The following table 5.2 shows a list of implemented events between server and client.

event	*description*
info	the server is sending information about its status
execute	a signal to the client that it should execute all the tests

Table 5.2: Events emitted by server

5.4 Caching - optimizing for speed

The testing and development system needs to be fast and reliable. In this paragraph I will describe one interesting implementational detail - heavy caching.

When optimizing for speed, it is important to realize where the real bottlenecks are instead of micro optimizing. In our case, there are three activities where most of the time is being spent:

- the actual JavaScript execution in the web browser

- network transport
- file access

Improving the speed of JavaScript execution in the browser is very difficult for us to effect. It is more about the developer and the way he writes the code. This topic would be a separate paper - separating logic from the presentation to minimize testing with DOM, not appending DOM objects into document to avoid rendering or faking asynchronicity to avoid context switching between JavaScript and C/C++. Also proper memory management, as memory leaks can slow down the execution significantly. All of these are out of the scope of this thesis.

Minimizing network operations and file access is however possible. We do that through heavy caching in the client.

The server maintains FS Model which has timestamps of last change of all source files[4]. When the web server is generating the `context.html` (which is the context page for test execution that is reloaded in the iframe and contains list of all the files to load), it includes `<script>` tags using these timestamps. For instance, when loading a file `/src/some.js`, it will include `<script src="/src/some.js?123456"></script>`, where `123456` is the timestamp of the last change to that file. The web server, when serving that particular file, it sends headers for heavy caching, so that browser never ask for this file again and rather keep it in its in-memory cache. If the file changes, the `context.html` is reloaded and the file will be included with the new timestamp and therefore the client will fetch the file from server again.

This results in only loading the files when they actually changed, which saves a significant number of HTTP requests.

5.5 Batching multiple file changes

There is an interesting paradox we are hitting while designing a testing environment. We need to run the tests as fast as possible, but in a situation when we are for example updating the whole project via a source control management system, we need to introduce a small delay before starting the test. We need to find a reasonable compromise.

When multiple file changes happen at the same time - or within time interval of couple of hundreds of milliseconds[5], we do not want to trigger multiple test runs, we want to batch all the changes and run the tests only once.

This section describes the implementation of such a feature using promises.

5.5.1 Concept of Promise

A promise is an old concept that has been recently reinvented in JavaScript world. It is very powerful and clean way of dealing with asynchronous APIs.

[4]This should be a SHA of the content of the file, so that even if the file was edited without actually making any change, we do not fetch it again.

[5]This can typically happen when IDE saves all open files, or during git checkout

An asynchronous function immediately returns the execution to the caller, but without returning any value. Typically the last argument of such a function is a callback. This callback will be eventually called, once the operation is finished. Code 5.6 is an example of reading a file using such an asynchronous function.

```
// reading a file using asynchronous API
fs.readFile('/some/file.js', function(error, data) {
  console.log('callback');
});
console.log('done');

// produced output
done
callback
```

Code 5.6: Asynchronous API using a callback function

Code 5.7 is an example of the same API, but using a promise. The `fs.readFile` function immediately returns a value - a promise. The promise is a container, that will eventually contain the value (content of `/some/file.js` in this case). The advantage over the previous example is that we can immediately use this container - we can pass it around in between components.

```
var promisedFile = fs.readFile('/some/file.js');
// now I can pass promisedFile around to some other component

promisedFile.then(function(data) {
  console.log('callback');
});
```

Code 5.7: Asynchronous API using a promise

Note: This is only a very basic explanation of promises, just for the purpose of explaining batching multiple file changes in Karma. There are many other advantages of using promises.

5.5.2 Batching multiple file changes

Sequential diagram 5.5 describes the communication between all the participating components over the time. When a file is changed, FS Watcher updates FS Model ("file change" in the diagram) by calling either `removeFile()` or `addFile()` or `changeFile()`. FS model updates its state and emits `file_list_modified` event with a single argument - a promise. This promise will eventually contain all the files, but at this point it is not resolved (does not contain the value) yet. There are two components listening on `file_list_modified` event - Manager and Web Server. Web Server only stores this promise for the future. Manager checks if all the captured clients are ready for test execution and triggers test execution by sending a message to all captured clients ("trigger execution" in the diagram). From the client point of view, test execution is just reloading the iframe with the latest context.html, so it sends an HTTP request to Web Server. At that point, Web Server is waiting for resolving the promise - delaying the response.

Since the first file change, FS Model waits for configured time interval (the default value is 250ms) and then resolves the promise. All subsequent file changes that happen within this time frame are ignored - FS Model only updates its state, but does not emit `file_list_modified` event again. When the promise is resolved (that means it contains the list of all source files with their last modified timestamps), Web Server finally responds the delayed request.

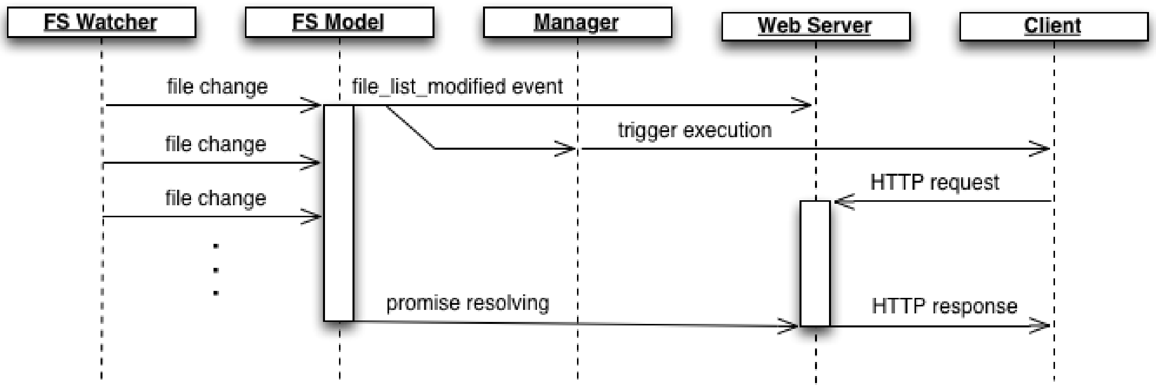


Figure 5.5: Batching multiple changes using promise

Sequential diagram 5.6 shows a simpler implementation, where FS Model delays emitting `file_list_modified` event. The advantage of our implementation described in diagram 5.5 is that the trigger signal to the client ("trigger execution" in the diagram) and the HTTP request from the client to Web Server are done in parallel to the delaying. In diagram 5.6, these happen sequentially, which means an overhead of about 30ms.

5.6 Dependency Injection

Karma is implemented by following the principles of Inversion of Control (IoC) pattern and all the server side components are wired together by a Dependency Injection (DI) framework that I wrote for the purpose of Karma. In this section I briefly explain both the pattern and the framework and why I decided to use it.

5.6.1 Inversion of Control Pattern

IoC is about **separating instantiation of objects from the actual logic and behavior** that they encapsulate. Code 5.8 is an example of breaking this principle - `Browser`

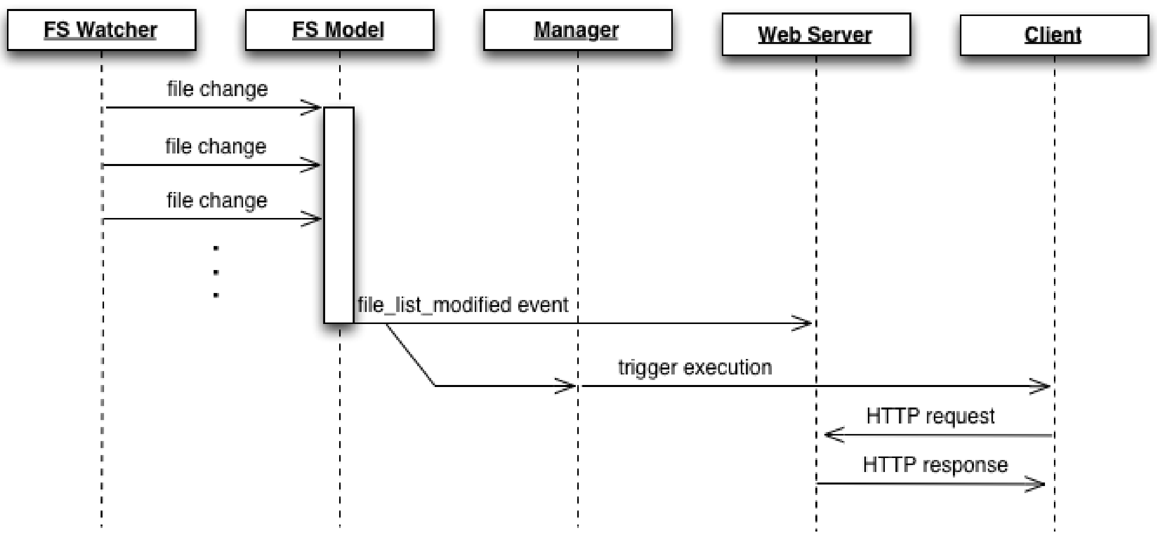


Figure 5.6: Simplified batching multiple changes

constructor does not have any argument, but when you try to instantiate it, it will instantiate its dependencies as well. It knows HOW to instantiate its dependencies and where to find them.

This also limits modularization and code reuse, because `Browser` is strongly coupled to the rest of the environment. It is difficult to instantiate `Browser` alone, without the rest of the system.

```
var Browser = function() {
  var emitter = events.getGlobalEmitter();
  var collection = new BrowserCollection(emitter);

  this.id = Browser.generateUID();

  // logic and behavior
  this.onRegister = function(info) {
    // ...
  };
};
```

Code 5.8: Example of mixing object instantiation and actual logic

Code 5.9 shows the same example but following IoC principle - `Browser` is not responsible for instantiating its dependencies anymore. It only declares them and it is responsibility of the creator to satisfy them. This is very important because now `Browser` is only coupled to particular interface[6].

```
var Browser = function(id, emitter, collection) {
  this.id = id;

  // logic and behavior
  this.onRegister = function(info) {
    // ...
  };
};
```

Code 5.9: Example of following IoC principle

This improves modularization and code reuse - it is easier to reuse `Browser` implementation in other environment as it is only coupled to interfaces rather than concrete implementations. Therefore we can use different implementations to satisfy these dependencies.

This loose coupling is essential for testing, because testable code is really about ability to instantiate an object stand-alone without the rest of the environment, usually using mock implementations of expensive dependencies. Testing was the core reason why I decided to follow the IoC principle since the very beginning. Development of this project is heavily based on testing and following the IoC pattern is the easiest way to write testable code.

5.6.2 Dependency Injection Framework

For a long time, I did not use any DI framework - all the assembling code was written manually. During the refactoring to enable extensibility via custom plugins, I realized that using DI framework would make it much easier and so I wrote a simple implementation of DI framework for Node.js and refactored the whole project to use it.

When following the IoC pattern, you end up with a lot of boilerplate code - the actual instantiation of all the objects and wiring them together[7].

DI framework makes the assembling part declarative rather than imperative - each component declares its direct dependencies and the DI container (typically called "injector") resolves these dependencies automatically. That means the dependency graph is not hard coded in the imperative code and the whole system gets more extensible and flexible. It is easier to assemble different system on different environments and also refactoring becomes easier.

[6]In JavaScript there is no notion of interface. It is only based on a convention - the name of the argument stands for a set of methods and properties that the interface have.

[7]Without IoC you would write the same code as well, but it would be spreaded across the codebase, mixed with the logic.

Chapter 6

Testing

In this chapter, I describe how Karma itself is tested, using testing on multiple levels. I describe which tools and libraries are used for testing and how it impacts the whole development workflow and open source collaboration.

The entire project has been developed using Test Driven Development practices since the very beginning. There are multiple levels of tests (∼250 unit tests for the server side code, ∼35 unit tests for the client side code and 12 end to end tests that test Karma as it is).

In the following sections, I describe different levels on which we test. Later, I describe some of the tools that we use.

6.1 Server Unit Tests

Each file in `lib/*` has a related test file in `test/unit/*`. These tests are written in CoffeeScript [6]. CoffeeScript has very clean and terse syntax, but debugging is more difficult, as you have to debug compiled JavaScript. That is why I decided to only use CoffeeScript for writing tests, not the production code. The main reason for using CoffeeScript at all was that I wanted to try it and get some real experiences using it.

The server code only runs on Node. That is why we use Mocha [17] as the testing framework as well as the test runner.

Originally, server unit tests were written in Jasmine [38] and run using jasmine-node [16]. The main motivation for rewriting them into Mocha was a better test runner (Mocha is better runner than jasmine-node) and better syntax for asynchronous testing. As a testing framework, I like Jasmine better than Mocha[1]. Its syntax is very clean, descriptive and yet powerful. I did not realize that before - these are very subtle details that make Jasmine superior to anything else out there.

Server unit tests can be run by executing `grunt test:unit` from the command line.

[1]Mocha does not contain any assertion library neither mock library. Therefore we ended up using chai [1] for assertions and sinon [35] for mocking. These are both very powerful libraries, but their APIs are huge and therefore there are usually multiple ways of doing the same thing and it is also hard to remember.

6.2 Client Unit Tests

Unit tests for the client code are located in `test/client/*`. These tests are written using Jasmine as the testing framework. This code runs in browsers and therefore needs to be tested on real browsers. Of course, we use Karma itself to run these tests.

Client unit tests can be run by executing `grunt test:client` from the command line.

A common problem is how to get access to internals that are not exposed in production code. Code 6.1 is an example of such a code. The Karma constructor function is not a public API, but we want to have access to it during testing, so that we can instantiate different instances passing in mock dependencies. Also having access to internal functions allows us to test these functions directly, through very focused low level tests.

At the same time, we only want to expose the public API. The global state in the browser is shared between all scripts and therefore it is not a good practice to pollute the global state.

The solution that we are using here is "wrapping the code into a function closure". During development, all internal functions and variables are exposed as globals, so that we can access them during testing. During build process, the whole source code is wrapped into an anonymous function call, which is immediately executed and publishes only the public API. Code 6.2 shows the previous code as it looks when built - it is wrapped into an anonymous function call and only exposes a single instance of Karma, which is the public API.

The client code can be built by executing `grunt build` from the command line.

```
var Karma = function(socket, context, navigator) {
  this.result = function() {};
  this.complete = function() {};
  this.info = function() {};
  // ...
};
```

Code 6.1: Karma client API exposed for testing

6.3 End to End Tests

End to end tests are high level tests that test Karma as it is. These tests are located in `test/e2e/*`. Each directory is a self contained project and the test basically test that project using Karma.

In addition to actual testing, these e2e tests serve as complete examples on how to use Karma. For instance the "mocha" e2e test is basically a trivial project using Karma with Mocha testing framework, or "requirejs" e2e test is a simple project using Karma with Require.js framework.

End to end tests can be run by executing `grunt test:e2e` from the command line.

```
(function(window, document, io) {
 var Karma = function(socket, context) {
   this.result = function() {};
   this.complete = function() {};
   this.info = function() {};
   // ...
 };

 var socket = io.connect('http://' + location.host);

 // explicitly exposed to global window - public API
 window.karma = new Karma(socket,
                             document.getElementById('context'));ji
})(window, document, window.io);
```

Code 6.2: Wrapped into an anonymous function call

6.4 Travis CI

Travis [5] is a Continuous Integration (CI) server that integrates really well with GitHub. [9]. Every time any developer pushes new changes to GitHub, Travis checks out these latest changes and executes predefined tasks. In our case, that means building the client code, running all tests, linting all code and validating commit messages.

This is a huge help during development, as I do not have to run all the tests all the time. In fact, usually I only run unit tests and then push the changes to GitHub. Travis checks out my code and runs all the tests on two latest stable versions of Node. In the meantime, I can be working on something else. It also means that I do not have to rely on other developers to run the tests - even if they forget to do that, Travis will.

Travis integrates very well with GitHub - the configuration is very simple. It is a YAML file that specifies what versions of Node we want to run the tests on and what is the command to execute. Code 6.3 shows Travis configuration for Karma project.

When collaborating on an open source project, sending a pull request is the most typical workflow. The developer who wants to submit a patch makes changes in his own fork. Then, he sends a pull request - a request to pull in his changes. I need to review the changes he is proposing and either merge it or reject. At that point, it is also very helpful to know, if all the tests are still passing after merging this pull request. That is exactly what Travis does. It tries to merge these changes, runs all the tests and reports the results. Figure 6.1 shows a screenshot of such a pull request. The green bar showing "Good to merge" means that all the tests passed on Travis.

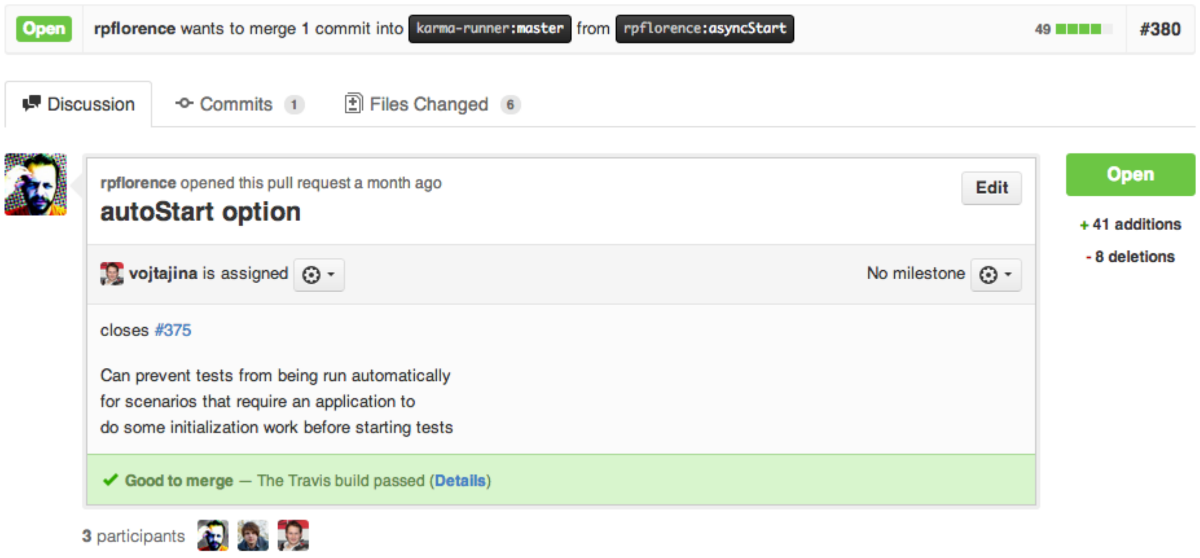
In our case, Travis build not only runs the tests but it also lints source files to make sure they follow our coding style and validates commit messages to follow our git commit message convention. This is a lot of manual work that I do not have to do, because Travis does it for me.

```
language: node_js
node_js:
 - 0.8
 - 0.10

before_script:
 - export DISPLAY=:99.0
 - sh -e /etc/init.d/xvfb start
 - npm install -g grunt-cli

script:
 - grunt
```

Code 6.3: Travis configuration



Open rpflorence wants to merge 1 commit into karma-runner:master from rpflorence:asyncStart 49 #380

Discussion Commits 1 Files Changed 6

rpflorence opened this pull request a month ago

autoStart option

vojtajina is assigned No milestone

closes #375

Can prevent tests from being run automatically
for scenarios that require an application to
do some initialization work before starting tests

✓ Good to merge — The Travis build passed (Details)

3 participants

Open
+ 41 additions
- 8 deletions

Figure 6.1: Travis screening a pull request

Chapter 7

Conclusion

In this last chapter, I summarize the whole project. Achievements, such as who is using Karma and some of the favourite quotes. I also suggest possible improvements and share my future plans with the project.

Overall, I am very satisfied with results of this project. The main reason is Karma itself - I feel confident to use it everyday and it significantly improved my daily workflow. The whole AngularJS team has been successfully using it as well - both for local development and on the CI server [55]. I think the fact that our team has been using it the whole time and I have been basically developing it for us, is the main reason why Karma provides such a good workflow.

At this point, many projects at Google and YouTube are using Karma[1]. There are also other big companies such as Spotify or LinkedIn that are using Karma.

There is no way to know the number of actual users, but the NPM web site [26][27] shows about 37k of downloads during the last month[2]. That said, I assume huge part of Karma users to be AngularJS developers. AngularJS has become very popular recently and we recommend Karma as "the test runner". That is probably one of the main reasons for its popularity.

During the development of Karma, I also created a few other small projects, such as:

- **mocks** [25] - a set of mocks for easier Node testing,
- **di** [18] - Dependency Injection framework for Node,
- plugins for Sublime Text editor such as OpenRelated [29] and BuildSwitcher [30],
- plugins for Grunt, such as grunt-coffee-lint [20] and grunt-bump [19].

[1]There are 14 projects that explicitly define dependency on Karma, however there are also many other projects using Karma outside of that.

[2]The number is a sum of two packages - "testacular" and "karma", because the project has been recently renamed and therefore both packages refer to the same project. These numbers unfortunately do not have any high value, as they do not mean number of actual users - a single user can easily download the package multiple times

I also wrote two blog posts for "howtonode.org", explaining techniques that I found useful during the Karma development [33][24]. These posts also inspired other people to create modules such as injectr [39] or rewire [34].

Recently, I wrote a blog post for Google Testing Blog [14] and did a couple of talks at conferences [22][28][21].

I recorded a screencast showing the basic setup and workflow [31]. Currently, the video has over 42k views.

All of the mentioned above taught me a lot! I learned a lot about Node.js and asynchronous programming. Even writing blog posts, recording the screencast or giving talks was a great experience for me.

Besides the project itself, something else - that is worthy mentioning - happened. Karma has been developed as an open source project at GitHub and that resulted into a great community of developers. At this point, there is ∼250 members on the mailing list group, with an average of about 150 posts a month. There is over 500 issues on GitHub with most of them already resolved. There is 180 pull requests with most of them already merged. These numbers are increasing every day and are showing that there is an active community of developers behind the project.

This is again something that taught me a lots - leading an open source project and the community behind it. That means a lot of decisions to make, as well as helping other people to understand reasons behind them.

7.1 Quotes

In this section I mention some nice quotes about Karma, mostly from Twitter or email conversations. I am mentioning it here, because I think it is a prove of developers finding Karma to be helpful.

"Karma has changed our life, now we test-drive everything."

— Matias Cudich (Tech Lead, YouTube)

"When we moved from JsTD, tests that took 1-3 minutes to run, we had about 150 unit tests (from 8-10 months of development). 4 months later, we have over 750 unit tests and they run in under 3 seconds every time we save a file. While there is still a lot of room for improvement in our own application, this test runner completely changed how our day-to-day work feels."

— Luke Bayes (YouTube)

"Big thanks to Julie for getting Karma setup, to be honest, this is the first time after joining Google, I enjoyed writing tests."

— some developer from Julie's team (Google)

"This shit is good!"

— Tyler Breisch (YouTube)

"Great news! Vojta (and all contributors!) thank you so much for this tool. The funny thing is that yesterday I was working on a native typeahead directive for AngularJS doing it TDD style and I was thinking to myself: it is so good that we've got Karma! Basically the testing experience is comparable to what I have in Java - and often better. So once again, congratulations and thnx for the awesome work. You are making solid testing practices possible in JavaScript world. For real!"

— Pawel Kozlowski (contributor to AngularJS)

"Coolest thing I've seen in a long time! Testacular - Spectacular Test Runner for JavaScript."

— Andrew Waterman @awaterma

"Wow, Testacular is a really great way to test JS in lots of browsers simultaneously."

— Andy Appleton @appltn

"This is the easiest browser testing I've tried. Testacular - Spectacular Test Runner for JavaScript."

— Matt Roman @mattroman

"Testacular is exactly what I wanted. Realtime automatic Jasmine test running with stupid easy config."

— Joel Hooks @jhooks

7.2 Suggested Improvements

Karma has already proven itself to be a helpful tool, but there is still a lot of space for improvements. In this section I briefly describe some of the possible improvements.

7.2.1 More efficient file watching

Chokidar, the library that Karma uses for watching the file system, uses Node's `fs.watchFile` API which in most of the cases does stat polling[3]. On huge projects with thousands of files, this can consume a significant amount of CPU resources.

I would like to improve the chokidar library to use `fs.watch` as much as possible, especially on Linux. We might also try a hybrid approach with combination of `fs.watch` and stat polling, because with fs.watch we can easily hit the OS limit for number of opened file descriptors.

[3]stat polling means it periodically stat the file system

7.2.2 Loading Only Required Files

Karma supports running only a subset of all the tests[4]. However, this filtering happens in the browser - the browser has to load, parse and evaluate all the files first. This can easily take a few seconds. If Karma collaborated with some dependency management framework such as Google Closure library [12] or Require.js [8], we could load only the files that are necessary for running given subset of tests. That would speed up the workflow significantly.

7.2.3 Collecting Usage Statistics

Karma is being used by many users, but we do not have any exact numbers. I would like to collect anonymous user data. That would help us understand the users better and make better decisions based on that knowledge. For instance, we could better design the APIs to have defaults that fit most of the users. We would also get better understanding of what features are actually being used and focus on improving them.

7.2.4 Integration with Browser Providers in the Cloud

There are services like BrowserStack [7] or SauceLabs [44] that provide browsers for testing. It would be very convenient if we had plugins for launching these remote browsers. Then, the developer could easily run the tests on almost any browser and any device, without actually having it.

7.2.5 Parallel Execution

Nowadays computers and even phones and tablets have typically multiple CPU cores. In order to make testing faster, we should take advantage of that. Karma itself is written in Node in a very non blocking way, which means that using multiple processes would not give us any major benefits[5].

We can however speed up the actual execution in the browser - by splitting the tests into smaller chunks and executing them in multiple browsers in parallel. Either on a single machine or using multiple machines.

[4]In fact, this is a feature of testing framework - Mocha and Jasmine

[5]This is one of the biggest advantages of Node.js. For more information on how Node event loop works, see [45].

7.3 Future of the Project

I would like to implement the improvements mentioned in the previous section and lead the project into a stable version. I plan to focus more on supporting other contributors and making sure that everybody understands the core values of Karma, which is speed, user experience and seamless workflow.

Then, I would like to pass most of the responsibility on the core community contributors, so that the project development can successfully continue even without me.

Bibliography

[1] Chai Assertion Library.
<http://chaijs.com/> .

[2] LiveScript - a language which compiles to JavaScript.
<http://livescript.net/> .

[3] Selenium - Web Browser Automation.
<http://docs.seleniumhq.org/> .

[4] Selenium WebDriver - Documentation.
<http://docs.seleniumhq.org/docs/03_webdriver.jsp> .

[5] Travis CI.
<https://travis-ci.org/> .

[6] Jeremy Ashkenas. CoffeeScript.
<http://coffeescript.org/> .

[7] BrowserStack. Live Web-Based Browser Testing.
<http://www.browserstack.com/> .

[8] James Burke. A JavaScript Module Loader.
<http://requirejs.org/> .

[9] GitHub, Inc. GitHub Homepage.
<https://github.com/> .

[10] GitHub, Inc. Top Languages.
<https://github.com/languages> .

[11] Google, Inc. AngularJS: Html Enhanced for Web Apps.
<http://angularjs.org> .

[12] Google, Inc. Closure Tools.
<https://developers.google.com/closure/library/> .

[13] Google, Inc. Dart: Structured web apps.
<http://www.dartlang.org/> .

[14] Google, Inc. Google Testing Blog: Testacular.
<http://googletesting.blogspot.com/2012/11/testacular-spectacular-test-runner-for.html> .

[15] Google, Inc. JsTestDriver - Remote JavaScript console.
<https://code.google.com/p/js-test-driver/> .

[16] Misko Hevery. Integration of Jasmine with Node.js.
<https://github.com/mhevery/jasmine-node> .

[17] TJ Holowaychuk. Mocha - the fun, simple, flexible JavaScript test framework.
<http://visionmedia.github.io/mocha/> .

[18] Vojta Jína. Dependency Injection for Node.js.
<https://github.com/vojtajina/node-di> .

[19] Vojta Jína. Grunt.js plugin - increment package version.
<https://github.com/vojtajina/grunt-bump> .

[20] Vojta Jína. Grunt.js plugin - lint CoffeeScript.
<https://github.com/vojtajina/grunt-coffee-lint> .

[21] Vojta Jína. GTAC 2013 - Karma.
<https://www.youtube.com/watch?feature=player_detailpage&v=yx6ErjPYDeY#t=19053s> .

[22] Vojta Jína. JS.Everywhere 2012 - Testacular.
<http://www.youtube.com/watch?v=5mHjJ4xf_K0> .

[23] Vojta Jína. Karma - Spectacular Test Runner for JavaScript.
<http://karma-runner.github.io/> .

[24] Vojta Jína. Make Your Tests Deterministic.
<http://howtonode.org/make-your-tests-deterministic> .

[25] Vojta Jína. Mocking library for Node.js.
<https://github.com/vojtajina/node-mocks> .

[26] Vojta Jína. NPM - karma package.
<https://npmjs.org/package/karma> .

[27] Vojta Jína. NPM - testacular package.
<https://npmjs.org/package/testacular> .

[28] Vojta Jína. PragueJS - Testacular.
<http://www.youtube.com/watch?v=xFD9bGP14Ww> .

[29] Vojta Jína. Sublime Text plugin for opening related files.
<https://github.com/vojtajina/sublime-OpenRelated> .

[30] Vojta Jína. Sublime Text plugin for switching build systems.
<https://github.com/vojtajina/sublime-BuildSwitcher> .

[31] Vojta Jína. Testacular (now Karma) - JavaScript Test Runner.
<http://www.youtube.com/watch?v=MVw8N3hTfCI> .

[32] Vojta Jína. Testing Angular Directives - the easy way.
<https://github.com/vojtajina/ng-directive-testing> .

[33] Vojta Jína. Testing Private State and Mocking Dependencies.
<http://howtonode.org/testing-private-state-and-mocking-deps> .

[34] Johannes. Dependency injection for node.js.
<https://github.com/jhnns/rewire> .

[35] Christian Johansen. Sinon.JS.
<http://sinonjs.org/> .

[36] Joyent, Inc. Node.js.
<http://nodejs.org> .

[37] The jQuery Foundation. QUnit: A JavaScript Unit Testing framework.
<http://qunitjs.com/> .

[38] Pivotal Labs. Jasmine - A JavaScript Testing Framework.
<http://pivotal.github.io/jasmine/> .

[39] Nathan MacInnes. Dependency injection for node.js.
<https://github.com/nathanmacinnes/injectr> .

[40] Microsoft. TypeScript.
<http://www.typescriptlang.org/> .

[41] Paul Miller. Chokidar.
<https://github.com/paulmillr/chokidar> .

[42] NodeJitsu. A full-featured http proxy for Node.js.
<https://github.com/nodejitsu/node-http-proxy> .

[43] Guillermo Rauch. Socket.IO - the cross-browser WebSocket for realtime apps.
<http://socket.io/> .

[44] Sauce Labs, Inc. Automated or Manual Cross Browser Testing.
<https://saucelabs.com/> .

[45] Mikito Takada. Understanding the node.js event loop.
<http://blog.mixu.net/2011/02/01/understanding-the-node-js-event-loop/>
.

[46] Bret Victor. Inventing on Principle.
<http://www.youtube.com/watch?v=PUv66718DII> .

[47] W3C. WebDriver.
<`http://www.w3.org/TR/webdriver/`> .

[48] Wikipedia. Client-server model.
<`https://en.wikipedia.org/wiki/Client-server_model`> .

[49] Wikipedia. Firefox OS.
<`http://en.wikipedia.org/wiki/Firefox_OS`> .

[50] Wikipedia. GitHub.
<`http://en.wikipedia.org/wiki/GitHub`> .

[51] Wikipedia. Google Chrome OS.
<`http://en.wikipedia.org/wiki/Google_Chrome_OS`> .

[52] Wikipedia. inotify.
<`http://en.wikipedia.org/wiki/Inotify`> .

[53] Wikipedia. Kqueue.
<`http://en.wikipedia.org/wiki/Kqueue`> .

[54] Wikipedia. Rhino (JavaScript Engine).
<`http://en.wikipedia.org/wiki/Rhino_(JavaScript_engine)`> .

[55] Alex Young. Totally Testacular.
<`http://dailyjs.com/2012/10/18/testacular/`> .

Appendix A

Acronyms

API Application Programming Interface

CI Continuous Integration

DI Dependency Injection

DOM Document Object Model

DSL Domain Specific Language

FS File System

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

IDE Integrated Development Environment

IoC Inversion of Control

JS JavaScript

SONP Spojené Ocelárny Národní Podnik Kladno

JSON JavaScript Object Notation

JSONP JSON with padding

MVC Model View Controller

NPM Node Package Manager

NVM Node Version Manager

OS Operating System

TDD Test Driven Development

UI User Interface

XML Extended Markup Language

Appendix B

User Manual

Full documentation can be found at Karma website [23].

B.1 Installation

Karma runs on Node.js and is available as a node module via NPM.

B.1.1 Requirements

First, you need to install Node.js. There are installers for both Mac and Windows. On Linux, we recommend using NVM.

B.1.2 Global Installation

This is the recommended way. It will install Karma into your global `node_modules` directory and create a symlink to its binary.

```
$ npm install -g karma

# start Karma
$ karma start
```

B.1.3 Local Installation

A local installation will install Karma into your current directory's `node_modules`. That allows you to have different versions for different projects.

```
$ npm install karma

# start Karma
$ ./node_modules/.bin/karma start
```

B.2 Configuration

In order to serve you well, Karma needs to know about your project. That's done through a configuration file. For an example file, see `test/client/karma.conf.js` which contains most of the options.

B.2.1 Generating the config file

You can write the config file by hand or copy paste it from another project. To make it simple, you can use `karma init` to generate it.

```
# This will ask you a few questions and generate a new config file
# called my.conf.js
$ karma init my.conf.js
```

B.2.2 Starting Karma

When starting Karma, you can pass a path to the configuration file as an argument. By default, Karma will look for `karma.conf.js` in the current directory.

```
# Start Karma using your configuration
$ karma start my.conf.js
```

B.2.3 Command line arguments

Some of the configurations can be specified as a command line argument, which overrides the configuration from the config file. Try `karma start --help` if you want to see all available options.

B.2.4 List of all config options

This is a list of all available config options, as well as their command line equivalents.

autoWatch

Type:	`Boolean`
Default:	`false`
CLI:	`-auto-watch`, `-no-auto-watch`
Description:	Enable or disable watching files and executing the tests whenever one of these files changes.

basePath

Type:	`String`
Default:	''
Description:	Base path, that will be used to resolve all relative paths defined in files and exclude. If `basePath` is a relative path, it will be resolved to the `__dirname` of the configuration file.

browsers

Type:	`Array`
Default:	`[]`
CLI:	`-browsers Chrome,Firefox`
Possible Values:	`Chrome`, `ChromeCanary`, `Firefox`, `Opera`, `Safari`, `PhantomJS`
Description:	A list of browsers to launch and capture. Once Karma is shut down, it will shut down these browsers as well. You can capture any browser manually just by opening a url, where Karma's web server is listening.

captureTimeout

Type:	`Number`
Default:	`60000`
Description:	Timeout for capturing a browser (in ms). If any browser does not get captured within the timeout, Karma will kill it and try to launch it again. After three attempts to capture it, Karma will give up.

colors

Type:	`Boolean`
Default:	`true`
CLI:	`-colors`, `-no-colors`
Description:	Enable or disable colors in the output (reporters and logs).

exclude

Type:	`Array`
Default:	`[]`
Description:	List of files/patterns to exclude from loaded files.

files

Type:	`Array`
Default:	`[]`
Description:	List of files/patterns to load in the browser.

hostname

Type:	`String`
Default:	`'localhost'`
Description:	Hostname to be used when capturing browsers.

logLevel

Type:	`Constant`
Default:	`LOG_INFO`
CLI:	`-log-level debug`
Possible values:	`LOG_DISABLE`, `LOG_ERROR`, `LOG_WARN`, `LOG_INFO`, `LOG_DEBUG`
Description:	Level of logging.

loggers

Type:	`Array`
Default:	`[{type: 'console'}]`
Description:	A list of log appenders to be used. See the documentation for log4js for more information.

port

Type:	`Number`
Default:	`9876`
CLI:	`-port 9876`
Description:	The port where the webserver will be listening.

preprocessors

Type:	`Object`
Default:	`{'**/*.coffee': 'coffee'}`
Description:	A map of preprocessors to use.

proxies

Type:	`Object`
Default:	`{}`
Description:	A map of path-proxy pairs.

reportSlowerThan

Type:	`Number`
Default:	`0`
Description:	Karma will report all the tests that are slower than given time limit (in ms). This is disabled by default.

reporters

Type:	`Array`
Default:	`['progress']`
CLI:	`-reporters progress,growl`
Possible Values:	`dots`, `progress`, `junit`, `growl`, `coverage`
Description:	A list of reporters to use.

runnerPort

Type:	`Number`
Default:	`9100`
CLI:	`-runner-port 9100`
Description:	The port where the server will be listening. This is only used when you are using karma run.

singleRun

Type:	`Boolean`
Default:	`false`
CLI:	`-single-run`, `no-single-run`
Description:	Continuous Integration mode. If `true`, it captures browsers, runs tests and exits with `0` exit code (if all tests passed) or `1` exit code (if any test failed).

urlRoot

Type:	`String`
Default:	`'/'`
Description:	The base url, where Karma runs. All the Karma's urls get prefixed with the `urlRoot`. This is helpful when using proxies, as sometimes you might want to proxy a url that is already taken by Karma.

Appendix C

Content of attached CD

|-**example**	An example project, configured to use Karma.
|	
|-**linux**	Node.js and Karma for Linux and other systems.
|--bin	Put this directory into your PATH.
|--karma	Installed Karma, with all its dependencies.
|--nodejs	Node.js source code.
|	
|-**mac**	Node.js installer and Karma for Mac OS.
|--bin	Put this directory into your PATH.
|--karma	Installed Karma, with all its dependencies.
|--nodejs	Node.js installer for Mac.
|	
|-**source**	Karma full source code, from GitHub.
|	
|-**text**	The thesis text.
|--source	LaTeX source of the thesis.
|--thesis.pdf	The thesis - generated pdf.
|--thesis-print.pdf	The thesis - generated pdf for printing.
|	
|-**web**	The Karma homepage that includes documentation.
|	
|-**windows**	Node.js installer and Karma for Windows.
|--bin	Put this directory into your PATH.
|--karma	Installed Karma, with all its dependencies.
|--nodejs	Node.js source code.
|	
|-readme.txt	Instructions on how to install and start using Karma.
|-tree.txt	Content of the CD.