Final thesis

Designing and implementing an architecture for single-page applications in Javascript and HTML5

by

Jesper Petersson

LIU-IDA/LITH-EX-A–12/063–SE

November 19, 2012

Supervisors: Johannes Edelstam (Valtech), Massimiliano Raciti (LiU)

Examiner: Prof. Simin Nadjm-Tehrani

Abstract

A single-page application is a website that retrieves all needed components in one single page load. The intention is to get a user experience that reminds more of a native application rather than a website. Single-page applications written in Javascript are becoming more and more popular, but when the size of the applications grows the complexity is also increased. A good architecture or a suitable framework is therefore needed.

In the thesis was first a number of design patterns suitable for GUI-applications analyzed. Based on a compound of these design patterns, an architecture that targets single-page applications was designed. The architecture was designed to make applications easy to develop, test and maintain. Initial loading time, data synchronization and search engine optimizations were also important aspects that were considered. A framework based on the architecture was implemented, tested and compared against other frameworks available on the market.

The framework that was implemented was designed to be modular, supports routing and templates as well as a number of different drivers for API communication. The modules were designed with a variant of the MVC-pattern, where a presentation model was introduced between the controller and the view. This allows unit tests to bypass the user interface and instead communicate directly with the core of the application. After minification and compression, the size of the framework is only 14.7 kB including all its dependencies. This results in a low initial loading time.

Finally, a solution that solves the problems regarding search-engine optimizations is also presented. It is based on an API written in PhantomJS that executes Javascript so a static snapshot can be served to the crawlers. The solution is fast, scalable and easy to maintain.

Acknowledgments

First off I would like to thank my examiner Simin Nadjm-Tehrani and my supervisor Massimiliano Raciti for taking the time to examine and supervise me during my master thesis. Your feedback have been really helpful and encouraging.

The master thesis was carried out at Valtech in Stockholm. It has been a real pleasure to be surrounded by professionals in diverse areas during the thesis, it has really helped me to look at the thesis from different angles. A special thanks goes to my supervisor and Javascript-guru Johannes Edelstam, I'm really grateful for all the help and support!

Glossary

Term	**Description**
AJAX	Asynchronous JavaScript and XML, a technique used to let clients asynchronously do API-requests after the page has been rendered.
API	Application Programming Interface, an interface for a computer program.
Backbone.js	A lightweight Javascript framework.
CDN	Content Delivery Network, a network of servers that serves content with high availability and performance.
Crawler	A program that browses the web, used by search engines to index websites.
DOM	Document Object Model, a tree representation of the objects on a web page.
DOM-element	A node in the DOM-tree.
DOM-selector	A method of retrieving nodes from the DOM-tree.
Ember.js	A Javascript framework with two-way data-bindings.
GET parameter	A parameter sent as a GET request in HTTP.
HTML	Hypertext Markup Language, a markup language used to describe objects on a website
HTTP	Hypertext Transfer Protocol, an application protocol used to distribute websites in a network.
Hashbang	A name for the character sequence #!
jQuery	A popular Javascript library commonly used in web applications.
JSON	JavaScript Object Notation, a text-based standard to format data in a compact way.
MySQL	An open source relational database manager.
PHP	A server-side scripting language commonly used for web development.
PhantomJS	A headless Webkit browser.
REST	REpresentational State Transfer, a simple software architecture style for distributed systems. Commonly used for web services.
SEO	Search-Engine Optimization, when optimizing a website with the goal to get a higher page rank.
SOAP	Simple Object Access Protocol, a software architecture style for distributed systems.
SPA	Single-Page Application, a web application that fits all content on one single web page.
SPI	A synonym for SPA.
Truthy	A value that is evaluated as true in a logical context. In Javascript, any value except false, NaN, null, undefined, 0 and "".
URL	Uniform Resource Locator, a string that refers to an Internet resource.
XML	Extensible Markup Language, a text-based standard to format data.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For many years traditional web services have been built upon complex backend systems that serve static HTML-files to the users. When the Web 2.0-era came around, techniques of dynamic content loading such as Asynchronous JavaScript and XML (AJAX) became popular. These techniques allow the application to fetch data from the server without reloading the entire page. However, most of the page is still initially rendered on the server side. What we are seeing today is a change, now the backend acts as a simple API and instead puts a lot more responsibility on the client. The client is often built in Javascript to allow a flexible and interactive user interface. This type of web application, known as Single-Page Application (SPA) or Single-Page Interface (SPI), radically changes the entire architecture. When the number of components in the client grows the complexity is also increased. This makes a good structure or framework a crucial part of the system design, this becomes important since Javascript is an interpreted and loosely typed language which allows many possible faults to be introduced. Today some of the most popular websites are built as SPAs. A few examples are Gmail, Twitter and Foursquare.

With the traditional approach HTML-files are rendered with the current content of the database. When data is changed the page must be reloaded to force the HTML-files to be re-generated from scratch. Even for sites using AJAX to dynamically change parts of the page this interaction is the most common case. On the other hand, a SPA allows a more flexible and elegant way of dealing with data. Once the user has loaded an initial slim version of the site all the data can be fetched asynchronously. This has several advantages. First an initial page is now rendered faster, since a smaller amount of data is transferred to the user. This is the case since every request doesn't have to include the presentation of the data. It also allows data to only be fetched one single time since the client now can act as a small cache memory of the database. All of this flexibility leads to a more responsive interface and in the end to a better user experience. Unfortunately this flexibility also comes with a price, the frontend is now a lot more complex than before. When data is represented both at the client and at the backend side an interface is needed for the communication in between. This interface needs to be implemented, for purpose of fetching, inserting, updating and deleting data. Problems regarding synchronization between user sessions may also arise since the client now acts as a cache memory to the database. When a user modifies data this may affect other users that are online, i.e. concurrency problems can appear. The responsibility of synchronization between users is now put on the developer. All of this leads to a more complex design of the frontend.

When the frontend becomes bigger and more complex it also requires more memory to run. Since a user often only works with a particular part of a system at a time, it would be a waste of memory to have the entire application instantiated. A structure that allows the developer to decide what parts of the system to run would solve this problem. If the design of these parts of the system could also be re-used in other projects, then the efficiency of the development would be greatly increased.

Another important aspect of software development is maintenance of the code. In bigger projects often several developers are working collaboratively to build the frontend of a system. Code that is easy to read and understand become key aspects to achieve high efficiency and good quality. The use of design patterns becomes an important part of the system design. Unfortunately, it is a complex choice to make since there are many aspects to consider.

Some criticism has been raised against the performance of SPAs in terms of initial loading time [1]. Due to limitations of the HTTP architecture might the initial loading time be higher when using a Javascript frontend compared to the traditional approach. This is the case since the client first downloads the initial page and then executes its Javascript. When this is done the user can be routed to the corresponding page and its content can be loaded. This leads to a higher initial loading time compared to just downloading the plain page from the server. However, once the initial page has been loaded and rendered the Javascript frontend will be able to perform a lot better. Since the initial loading time is a key factor to whether a user is willing to visit the site or not [2], it has become a crucial aspect of Javascript-based frontends.

Another problem that is introduced when building a frontend in Javascript is if users visit the site and have disabled or cannot run Javascript, then they won't be able to see the content of the site. Even though only 2% of today's users have Javascript disabled [3] this still represents a problem. Crawlers from search engines represent for example a small amount of the total number of users but they are still very important to consider. If a crawler cannot see the content of the page it is impossible for it to analyze what the site is all about. This would lead to a poor page rank at the search engine. For many of today's websites this is extremely important.

The goal of the thesis is to come up with a system design of a framework for more lightweight single-page applications. The architecture shall be based on design patterns well suited for web frontends. To give a good view of how design patterns can be used in practice the thesis clarifies how they can be typically used in a web application. Based on the result of the system design the framework was implemented, allowing applications to get a high degree of testability as well as encouraging code re-usage between projects. The initial loading time of the page is considered as an important aspect of the framework and it was in all cases minimized. The thesis proposes also a solution to the problem of search engine optimization for single-page applications.

The thesis has been carried out at Valtech at their office in Stockholm. Valtech is a global consultancy firm with offices spread all around the world including Europe, Asia and North America. They deliver solutions for the web and mobile market with focus on high quality products and intuitive user interfaces. Their web interfaces often make use of Javascript to achieve a better user experience. Since Valtech's employees have been working with web development for years they have got a lot of experience when it comes to system design of web frontends. By performing interviews, their experience has been as a fundamental source for better understanding what problems are most common when developing Javascript applications.

By looking at design patterns suitable for graphical interfaces an abstract idea of how to form the system architecture was developed. This idea was then further refined to fit a structure that was more suitable for the web. Today there already exists a number of Javascript frameworks, these were analyzed to find different ways of approaching the problems. The majority of the existing frameworks are using techniques that are well known, well tested and have a wide browser support. However, the upcoming HTML5 draft contains new exciting techniques introducing new ways of approaching the problems. These techniques were taken into account when designing the architecture of the framework. During the design phase suitable design patterns were selected for the upcoming implementation of the framework. To be able to design the framework with a high degree of testability a test framework for Javascript was used. It was important to understand how testing typically is carried out and what testing utilities that are commonly used. Once the system design was ready the framework was implemented. To verify that the thesis fulfilled its requirements a simple test application was developed. This application was built upon the framework in Javascript. The purpose was to demonstrate how a developer could use the framework and how the most common problems are solved. Finally the initial loading time and testability were verified by performing a number of tests. The results were then be compared against other competitors.

The thesis is limited to only review design patterns that have a clear connection to frontend development, other design patterns were not considered. Since the thesis focuses on system design of Javascript frontends, aspects regarding the backend were neither included within the scope. The system design of the frontend is not limited to a certain type of backend, any language or design used in the backend shall be able to coop with the frontend as long as they have a common interface for communication. When developing the framework were support for older browsers on the market not taken into account, the framework can rather be used as a showcase of what is possible to do within a near future.

The rest of the thesis is structured as follows:

- Chapter two describes the technical background behind SPAs.
- Chapter three discusses design patterns commonly used in graphical interfaces.
- Chapter four presents the architecture behind the framework and how search engine optimization problems can be approached.
- Chapter five describes the implementation of the framework.
- Chapter six presents measurements of the framework regarding testability and initial loading time. It also includes comparisons with other frameworks on the market.
- The seventh and last chapter discusses when SPAs are suitable to use and how the future of the framework might look like.

Chapter 2

Technical background

2.1 Single-page applications

As previously described SPAs introduce new demands regarding architecture and structure on the client side. To understand how these problems can be approached it is important to understand the basics behind how a single-page application works. There is yet no formal definition that describes what a SPA is. Many developers have their own view of what it exactly means, but Ali Mesbah and Arie van Deursen [4] have stated a quite clear definition in their paper about SPAs:

> "The single-page web interface is composed of individual components which can be updated/replaced independently, so that the entire page does not need to be reloaded on each user action."

The definition includes a number of key attributes that help to define a SPA:

- **Web interface** - Used on the web, focuses on user interfaces.
- **Individual components** - It's divided into smaller components that coop with each other.
- **Updates and replaces** - A component can at any time be changed or replaced with another component.
- **Reloading** - The entire page is never reloaded even though new content may be loaded into some sections.
- **User actions** - The SPA is responsible for handling user actions such as input from mouse and keyboard.

When running a Javascript SPA the browser first makes a request to the web server. The web server will then respond with the Javascript client including the resources needed to run it. Once the client is transferred it will be initialized and it will be ready to be run. When the user interacts with the client, such as clicking on graphical elements, this may require new data to be fetched from the server. Instead of reloading the entire page the client will instead make a small request to the API on the web server. The API is simply an interface that allows the clients to communicate with the server in a data format that is easy to understand for both the client and server [5]. A typical flow of communication for a SPA can be seen in figure 2.1.

2.2 URLs, routes and states

The URL structure of a website is an fundamental aspect to consider. In order to provide linkable and shareable URLs it is important that the URL is unique for the application's state. If the same URL would be used for several states they would not be accessible directly from the address bar, which would become impractical if a user stores or shares the current URL. The Javascript application needs to be able to map a URL into a certain state, this

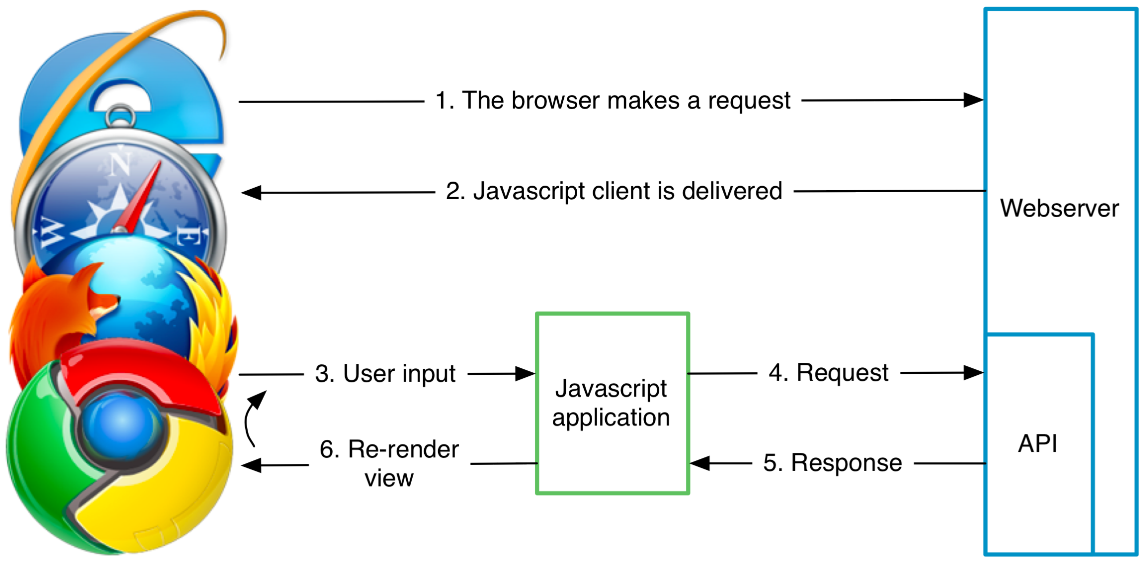


Figure 2.1: The flow of communication between a browser and the web server.

is often done by a routing component. It takes the URL that the user provides and decodes it into a state so that the correct components can be initialized and presented to the user. The routing component also needs to be able to update the URL if the application's state is changed. The HTML5 draft provides a history API to allow a Javascript client to change the URL during runtime without reloading the page [14]. To support older browsers that do not support the history API the hashbang technique can instead be used [15]. The idea is to add *#!/state* to the URL, the characters that comes after the hashmark (#) will not be sent to the server and can instead be used by the Javascript client to detect a certain state of the application.

Both the HTML5 history API and the hashbang technique will keep the back-button behavior intact. When the user hits the back-button in the browser the URL will simply change to the last visited URL. The Javascript application will recognize a change of URL and will load the content once again. By having a system like this it becomes possible to use URLs just as if the pages were rendered by a backend system instead by a Javascript application.

2.3 API communication

In order for the client to dynamically fetch data from the server an API is needed in between. The API describes how data is being transferred, what data format is being used and what methods are available.

One of the most common ways of letting the client asynchronously communicate with the server is via AJAX (Asynchronous Javascript and XML). AJAX allows the client to asynchronously fetch and post data to the server after the page has been loaded. However, AJAX has no native support for letting the server push data to the client. The client must always initialize a new connection and then poll the server for new data [16, p. 491]. To enable the server to push data to the client other techniques can instead be used, e.g. the websocket API in the HTML5 draft which supports bi-directional communication. This will ease the load of the server and allows a faster and more responsive system.

When using a HTTP-based data transportation some kind of architectural structure is needed. The architecture describes how messages are formed so that both parties understand each other, e.g. what methods that are available and how data is transferred. It also often includes additional information about how authentication is handled, as well as the flow of communication. Common architectural structures are REST and SOAP.

The data format of a message describes the syntactical form of the objects that are transferred. Since HTTP messages are based on ASCII-characters [20] it is also commonly used when describing data objects. Common data formats used today are JSON, XML and plain text [5].

2.4 Templates

In order to separate the presentation layer from the business logic, templates are often used. A template engine compiles HTML from a description written in a template language together with a set of data, it's commonly used for rendering the views in a Javascript application. When the data is changed the view can simply be re-rendered so that the user sees the correct version of it. A template engine is however a quite complex system, to generate HTML from a description an interpretation of the template language is required. Since these interpretations are done during runtime the performance is a crucial aspect to consider. Some frameworks have their own template engines while others use generic libraries in order to support template rendering.

2.5 Data bindings

Data bindings are used to bind a set of data to a corresponding view. One-way data bindings allow a view to be automatically re-rendered when its data is changed. Some frameworks, like Angular JS, support a two-way data bindings that also allow the data to be updated corresponding to changes in the view [21]. This is quite a powerful technique since all logic behind a view now is managed by the framework which makes it easier for the developer to get things working. Important to note is that no logic or complexity is removed, it is just moved to the framework. However, a framework must support a wide range of HTML-elements even though all of these are not used in the application. This often affects the performance of the application and two-way data bindings can sometimes become overkill, especially for smaller applications.

2.6 Testing a Javascript application

Testing comes in many forms and shapes during a project. During the development phase unit tests are commonly used to drive the development. It's used as a tool by the developers to ensure that all the functionality is kept intact. The use of a testdriven approach to develop software is becoming more and more popular [9]. However, developing Javascript applications with a testdriven approach can be tricky. The functionality behind a Javascript application tends to be tightly coupled with the user interface. This is simply the case since one of the most essential purposes of a Javascript applications is to interact with the user, which of course is done via the user interface. To write unit tests that interacts with the user interface is not trivial. It requires the unit test to trigger events on DOM-elements and then traverse the DOM and validate its content. If a view is even slightly changed this can break the test even if an acceptance test would still pass. Some architectures allow developers to instead write unit tests for what is underneath the graphical interface, making the tests easier to write and the result easier to validate. In the end unit testing will only ensure that the code is doing the right thing, not that the system is working as expected. High-level testing, such as acceptance testing, will always be needed to validate the functionality from a user perspective [10].

2.7 Search engine optimization

Search engine optimization is an important aspect of today's web services. Search engines such as Google, Bing and Yahoo indexes webpages every day to make it easier for users to find what they are looking for. In order to get a high page rank it is important that the search engines can index the content of the web service to understand what it is about.

The indexing is done by a crawler. The crawler downloads a snapshot of the webpage and analyzes the HTML to find keywords, links and more. Even though the crawler itself is quite a complex system the support for Javascript is limited. Google crawler won't run any Javascript at all [6], even though there are other reports stating that it actually can run a limited amount of Javascript [17]. Nevertheless, the limited support is a major issue when building Javascript-based sites that are to be indexed by a crawler.

In terms of the crawlers from Google, Bing and Yahoo is this solved by translating the URLs for AJAX-based services, this allows the webserver to serve static HTML snapshots instead of dynamic Javascript versions. When the crawler visits a website that has an URL that contains a hashbang (#!) the site will be treated as an AJAX-based site. All characters after the hashbang are then sent as the GET parameter *_escaped_fragment_* [18]. The reason for this is due to the HTTP specification stating that characters after a hashmark should not be included in the request to the webserver [19]. The webserver can check if the GET parameter *_escaped_fragment_* is set, if so is the client probably a crawler and a static HTML version is to be served. Figure 2.2 shows how a typical crawler treats AJAX-based websites.

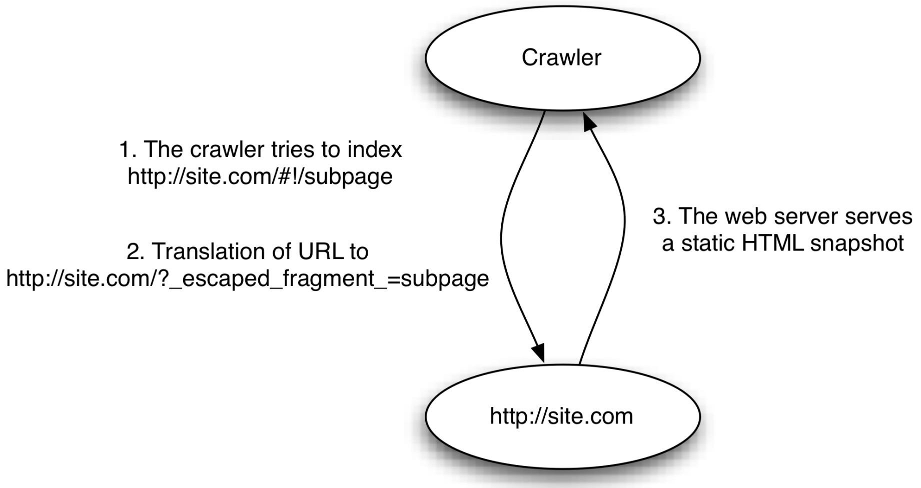

Crawler
1. The crawler tries to index
http://site.com/#!/subpage
2. Translation of URL to
http://site.com/?_escaped_fragment_=subpage
3. The web server serves
a static HTML snapshot
http://site.com

Figure 2.2: The flow of how a typical crawler communicates with an AJAX-based website.

Chapter 3

Design patterns

3.1 Modularity

Frameworks such as Sproutcore and Angular.js allow applications to be divided into smaller components [7] [8]. In bigger projects is modularity a key aspect to improve maintainability. If one component cannot be replaced without affecting the other components the application will easily become hard to manage in the long run. When each component solves a specific task they can also be developed in parallel, it also becomes easier to test and maintain the entire application [11]. A modular pattern is well suited since a website is naturally divided into several different pages where each page has its own functionality.

Another feature of a modular design is that it allows for code re-usage. If a module written in one project could be used in a new project without any major changes the efficiency would be improved. It also becomes easier to achieve high-quality code since already tested code can now be re-used.

3.1.1 The module pattern

By using a modular design pattern the architecture can be formed in a way that allows smaller components to be grouped together into a common module. The idea behind the module pattern is to provide private and public encapsulations, like normal classes. There are several ways of implementing this pattern in Javascript. However, important to note is that all these implementations only emulates traditional classes since Javascript doesn't support classes and access modifiers [12, p. 27]. Private members can however easily be emulated by hiding them in a function scope that only is available within the instance. An example of a module implementation in Javascript is described in listing 3.1.

```
function MyModule() {
  // Private variables
  var private_variable = 0;

  // Private methods
  var private_method = function() {
    return private_variable;
  };

  // Public variable
  this.public_variable = 10;

  // Public methods
  this.public_method = function() {
    return private_variable;
  };
}
```

Listing 3.1: A simple example of how to implement a module in Javascript using the function literal notation.

3.1.2 The singleton pattern

The singleton pattern restricts the number of instances of a class to one single instance. This is suitable for when writing components that are supposed to have a unique instance which is used across the entire system. This tends to be common in web applications, typical examples are API, messaging and debug components. Along with this the singleton pattern has the same advantages as the traditional module pattern.

3.1.3 Inheritance

Class inheritance is an important aspect of a modular pattern. In Javascript a prototype-based inheritance model is used [16, p. 199]. An example of how the prototype-based inheritance model works in Javascript is shown in listing 3.2.

```
// Inheritance implementation
function extend(Child, Parent) {
  // Let Child inherit from Parent
  Child.prototype = new Parent();
  Child.prototype.constructor = Child;
}

function Parent() {
  this.method1 = function() {
    return "Method 1 - Parent";
  }
  this.method2 = function() {
    return "Method 2 - Parent";
  }
}
function Child() {
  this.method1 = function() {
    return "Method 1 - Child";
  }
}
extend(Child, Parent);

var pi = new Parent();
pi.method1(); // "Method 1 - Parent"
pi.method2(); // "Method 2 - Parent"

var ci = new Child();
ci.method1(); // "Method 1 - Child"
ci.method2(); // "Method 2 - Parent"
```

Listing 3.2: Prototype-based inheritance in Javascript. Child inherits from Parent.

3.1.4 Decorating modules

There is sometimes a need for instantiating the same module with different amount of properties. When the number of properties grows subclassing will often become impractical since a new class for each combination of options needs to be created. The decorating pattern aims to solve this problem. It is commonly used by Javascript developers since it can be implemented transparently [12, p. 70]. The real value of using this pattern is the code re-usage. By letting a module be instantiated with options it will make it generic and can easily to be shared between projects.

3.2 Communication between modules

To provide a modular system some kind of interface is needed. In other languages like Java the purpose of interfaces is to describe what methods must be implemented so that other modules know what to expect. The advantage of using well defined interfaces is that a module becomes easy to use, self-documented and promotes code re-usability [12, p. 65]. The drawback is that Javascript doesn't come with any built in support for native interfaces, and since it is a loosely typed language some kind of validation is needed [12, p. 65]. The validation is simply needed in order to make sure that the correct type of data is sent to the module. Since this is done during runtime this will affect the performance of the system, in a compiled language this would not be a problem since it would be done during compilation.

Instead of directly calling a method on a class instance through an interface the observer pattern can be used. It allows a module to register on events from other modules. When something interesting happens the responsible module can inform its subscribers about it. This allows for a more flexible interface design and will make the modules more independent of each other [12, p. 38]. If one module fails to deliver a message the entire system will not crash, only the specific functionality that depended on that message will be affected.

Another possibility to get more loosely coupled modules is to use the command pattern as communication between the modules. Instead of calling methods directly on an instance every class will inherit a generic execution-function. This function will take a string as argument of what method to run. This protects the system from crashing if a method is called that is not implemented. For compiled languages this is not a relevant issue, but for interpreted languages such as Javascript there is no compiler that can help. An example how the command pattern can be implemented and used can be seen in listing 3.3.

```
// The CommandPattern prototype
function CommandPattern() {
  this.execute = function(method) {
    if(typeof(this[method]) === 'function') {
      return this[method].apply(this,
        [].slice.call(arguments, 1));
    }
    return false;
  }
};

// An example module
function MyModule() {
  this.myMethod = function(arg1, arg2) {
    console.log(arg1 + " " + arg2);
  };
};

// Let MyModule inherit from CommandPattern
MyModule.prototype = new CommandPattern();
MyModule.prototype.constructor = MyModule;

// Create an instance of MyModule
var instance = new MyModule();

// Run myMethod() on instance with foo and bar as arguments
instance.execute("myMethod", "foo", "bar");
```

Listing 3.3: An example of how the command pattern can be implemented and used in Javascript.

3.3 MVC - Model-View-Controller

MVC, Model-View-Controller, originated from the early days of Smalltalk-80 [22, p. 4]. The pattern is commonly used when creating user interfaces. The main idea behind MVC is to separate the presentation layer from the data layer. It's basically a pattern that can be constructed by combining the composite-, observer- and strategy-pattern all together [23]. MVC divides the design into models, view and controllers [22, p. 14]:

- Model - A record of data, completely ignores the UI
- View - Presents the model in a graphical form
- Controller - Takes user input and figures out what to do with it

The view observes the model for changes. If the model changes the view is re-rendered to keep the presentation up-to-date. On user input the controller will take a decision of what to do, if this involves changing a model it will tell the model to do so. The view will then be informed about this due to its observations. This means that the controller never really care what views are rendered. The view itself is composed by several UI-elements and implements therefore the composition-pattern. Figure 3.1 describes the communication between the different components of the MVC pattern.

Since MVC was designed with user interfaces in mind it also becomes a suitable choice when designing an architecture for a Javascript frontend. Most of todays Javascript-frameworks are based on MVC. Some of them are classical MVC-frameworks while others have made a few modifications.

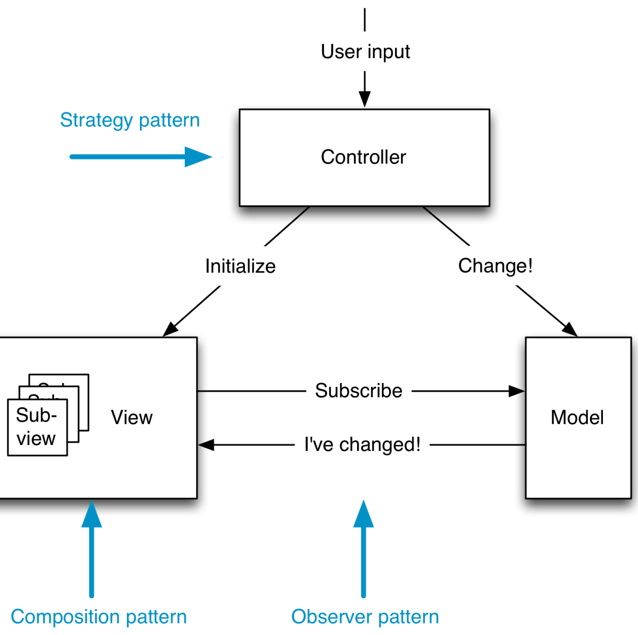


Figure 3.1: Communication between the different parts of the MVC pattern.

However, MVC has a few drawbacks as well. The view often has to keep track of several different UI-specific properties. The rest of the system is not interested in these UI properties, but the view cannot operate correctly without them. An example is if an image slideshow should change image periodically, the view needs to know which the current image is so that the next one can be loaded. However, this information is useless for the rest of the system, it is only needed by the view. A combination of the model and UI properties forms a state of what is currently presented to the user. Suddenly the view is supposed to take care of both presentation logic and UI elements. They can easily become quite complex and hard to maintain.

Also worth mentioning is MVP, Model-View-Presenter, which is a lot like the traditional MVC pattern. The main difference is that in MVP the user actions are captured by the view that raises an event, compared to MVC where the controller captures the user actions directly. By letting the view capture the user actions the coupling between the controller and the UI-elements will be reduced, improving the ability to re-use the code.

3.4 MVVM - Model-View-ViewModel

MVVM is a quite new compound of design patterns, originated from Microsoft as late as in 2005 [24]. It is currently used in Silverlight as well as in a number of Javascript frameworks such as KnockoutJS, Kendo MVVM and Knockback.js [25]. MVVM is designed to make two-way data-bindings easier to implement and use. It consists of three type of components: models, views and view-models. The views and models in MVVM have the same purpose as in the traditional MVC pattern. However, the view-model binds elements in the view to a specific model. Due to the two-way data binding the controller is no longer needed since changes in the view directly correspond to changes in the model and vice-versa. Other responsibilities that the controller had in MVC moved to the view-model, such as replacing views and deciding when to fetch data from the backend.

The main drawback of MVVM is that the view-model quickly grows and becomes complex to maintain since it has quite a lot of responsibilities [26]. Another drawback of the MVVM pattern is that it is designed to be used together with two-way data-bindings which easily can make it overkill for smaller systems with a relative few number of UI elements. Other patterns, like MVC or MVP, might in that case be a better alternative.

Chapter 4

Architecture

When creating the architecture of a system that is supposed to provide a high grade of code re-usage is modularity and well defined abstraction layers important. By using the module pattern and letting each module have well defined responsibilities the system will be easier to maintain in the long run. The goal is to be able to move a module and put it in another project without any major modifications. This would allow a module to be replaced later on if the requirements of the system have changed. For this to work it is important that the module can function by itself. If it must interact with other modules this must be done through well defined interfaces. Another important aspect to keep in mind is that the whole system shall not crash just because one module fails. In order to fulfill these goals the design will be divided into components where each component can either be constructed by other components or have a primitive task itself.

4.1 System-level architecture

The entire system can be divided into three separate components: a backend API, an API driver and the Javascript application itself. The backend API provides the Javascript application with data and the driver takes care of the abstraction in between. By utilizing a driver it becomes possible to replace the backend or frontend independently from each other. The user interacts with the Javascript application by either providing a URL to the system or in terms of user input such as clicking and pressing buttons. The application can at any time change the DOM (Document Object Model) to give the user feedback of the provided input actions. The DOM is the browser's representation of the elements on the page and is used when rendering it. An abstract view of the architecture seen from a system-level point of view can be seen in figure 4.1.

Many other frameworks do not have the API driver as a bridge between the backend API and the Javascript application, making them tightly coupled together. For example, to change the backend from using AJAX requests to HTML5 websockets would in most cases require a redesign of the entire Javascript application. By letting the driver have a well defined interface the components can be replaced independent of each other. It also allows drivers to be re-used between project since the interface is the same independently on the application.

4.2 Application-level architecture

The system architecture can be further refined. The Javascript application can itself be constructed by a number of other components including a router, a configuration as well as a number of modules and services. The router's task is to take the URL provided by the user and, depending on its value, load a specific module. This might imply that another module first has to be unloaded. The configuration provides an application-specific key-value-storage for global configuration properties. This is for example used to specify what API driver to use. The application-level architecture is described in figure 4.2.

The idea is that the application will only run one module at a time. When a module is loaded the already active module is unloaded and its memory is released. This makes the application more memory-efficient as well as requiring the developer to think in terms

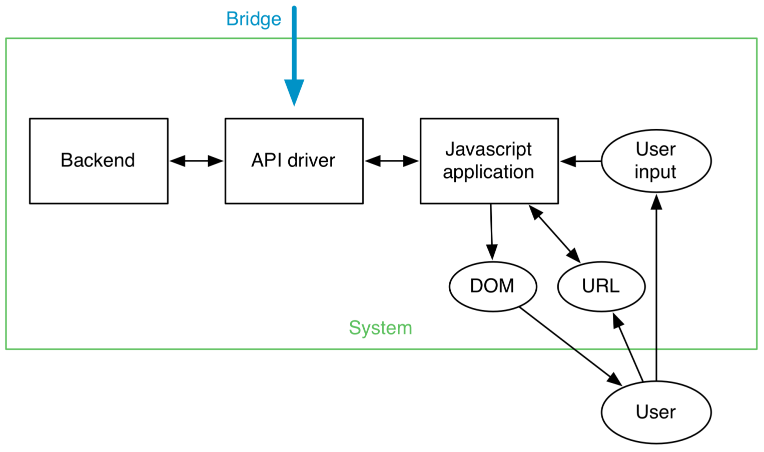


Figure 4.1: The architecture seen from a system-level point of view.

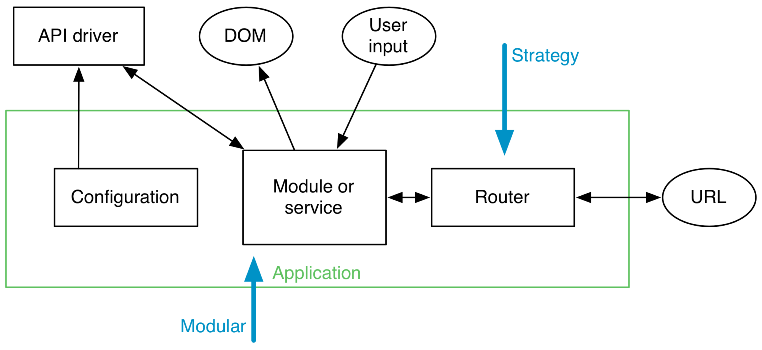


Figure 4.2: The architecture seen from an application-level.

of modules. Modules are supposed to deal with functionality that are page-specific for the site since each module typically implements a group of related features in the application. To deal with components that are used across the entire system services are introduced. A service is just like a module, except for one important difference. An application can run several services at a time where each service can be utilized by modules to perform common tasks in the system. Since the system only needs to run one single instance of each service the singleton pattern is used. This will ensure that the same instance is used across the whole system.

At this level the system can basically be seen as a number of modules that can be loaded and unloaded depending on the URL. It would now be possible to write a module for one project and then move it to another one later on. As long as the module can function by itself it will work there without any changes at all. This is why it is so important that a module can function by itself, all communication is done via a well defined interface which is the same independently on which application is running the module. However, a module can itself be quite complex, so a good module structure is still needed. The design must be further refined.

4.3 Module-level architecture

To refine the module structure even further its interface can be analyzed. A module will typically be required to handle data from the API, render HTML-views as well as take care of the user's input. Since the architecture targets smaller applications two-way data bindings would be overkill to use, if the framework would handle all needed bindings it would simply affect the performance too much. With this in mind the MVVM pattern is not really an alternative to use since it is especially designed for using two-way data bindings. A compound of patterns based on MVC would on the other hand work well. MVC was designed to get low coupling between components which of course is an important property to consider a modular system.

However, properties associated with a specific view are an issue with the traditional MVC pattern. In a Javascript implementation the view would represent an HTML-template, which definitely shouldn't be responsible for storing properties related to the presentation since this simply would mix logic and presentation too much. One alternative is to let the controller hold these properties. However, by doing so would make it responsible for several views and would tightly couple them together. Another solution to the problem is to introduce a presentation model in between the controller and the view. The presentation model handles all the UI-properties while the view can focus on rendering UI-elements. By letting the presentation model wrap all the model-objects into property-objects UI-specific properties can be added. This gives a better abstraction even though there now is another component to take into account. When using a presentation model the view now can be implemented with a template language, like Underscore.js or Handlebars.js, while the presentation model can be represented by a component that handles the state of the view. This would remove the dependency between the controller and the views and at the same time allow for a flexible way of dealing with UI-properties.

In traditional MVC the controller would be responsible for taking decisions based on the user input. However, letting the controller be responsible for listening on user input would become problematic in a web-based architecture. User actions are triggered on a DOM-element in the view. This would tightly couple the controller with the view and simply by-pass the presentation model. By instead going towards an MVP concept and let the presentation model handle the user input would keep the abstraction intact. Via a message channel the presentation model can report upwards to the controller that a certain action has occurred. By doing so a decision can be taken by the controller which is implementing the strategy pattern.

Another advantage of introducing the presentation model is that the testability is improved. This is the case since a unit test now can send a message to the controller via the message channel that a user has clicked a button. The unit test can then verify that the right actions are performed by looking at the data structure and see that it has changed. Most of the features of the application can therefore be tested during development with simple unit tests without interacting with the user interface at all.

By wrapping several models into collections and implementing the composite pattern operations can be performed on several models at a time. This can be quite useful since lists of data objects tend to be common in web applications. A presentation model can observe an entire collection of models to get informed whenever there is a change in a model. By utilizing collections operations such as sorting, filtering and pagination are also easy to implement and use.

In order to handle more complex views that might need to be divided into subviews is the composite pattern implemented. By using this pattern a view can be structured like a tree where each node can have other subviews as leafs. Since the DOM itself is represented by a tree-structure containing nodes of DOM-elements it is a highly feasible pattern to use for this purpose.

An illustration of the structure of a module can be seen in figure 4.3.

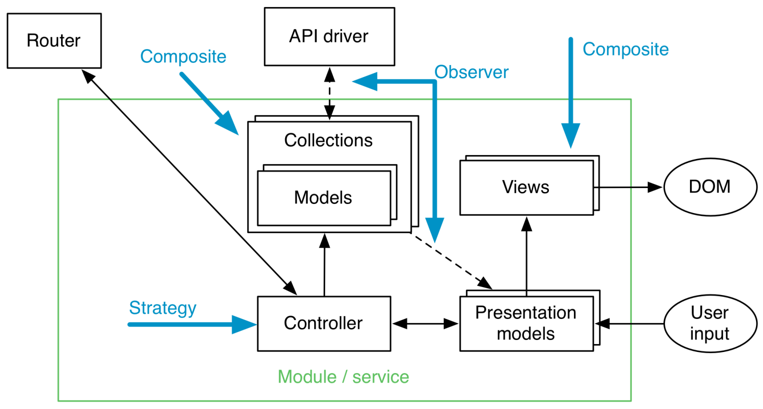


Figure 4.3: An abstract view of a module/service in the application.

4.4 API communication

In order to make the frontend independent on what backend is being used an API driver is introduced. The API driver works as a bridge between the Javascript application and the backend API. It is responsible for implementing a number of methods such as fetch, insert, update and remove. The API driver can be shared between projects since it is typically based on a number of API properties, as long as these properties are the same for two projects the driver will be compatible with both of them. Examples of properties are data transportation (REST, SOAP etc.), data format (JSON, XML etc.) and how the data is to be transferred (AJAX, websockets etc.). The main advantage with an API driver is that if one of these properties are changed a patch is only needed in the API driver, the entire application is not directly affected. For many other frameworks that are lacking support for API drivers it would typically be required to re-write major parts of the application to support the new set of API properties. Another advantage is that frontend developers can start developing the application before the final version of the API is finished. They can for example use an API driver for local storage during the initial development phase and then switch to a driver that is adapted for the API as soon as it is finished. This would allow the frontend to be developed in parallel with the backend. The drawback with having an API driver is some performance loss since it introduces another layer in the architecture. However, this delay can in most cases be neglected since it is really small compared to the time it takes to transfer the message to the server.

4.5 Initial loading time

As briefly mentioned in the introduction the initial loading time is a crucial metric to consider. The initial loading time is measured from the time the user has entered the URL in the browser until the page is rendered. Studies shows that 25% of the users close a page if it takes more than 4 seconds to load, and typically slow pages are less popular [2]. In order to minimize the initial loading time the process can be divided into a number of phases. For SPAs the phases are as follows:

1. A user enters the URL in the browser
2. The page is downloaded
3. The Javascript is downloaded

4. The Javascript is parsed
5. The application is initialized
6. A module is loaded
7. If data from the database is needed, it is fetched
8. The page is rendered

The flow for SPAs can be compared to the flow for traditional sites that only got phase 1, 2, and 8. Hence traditional websites will perform better. It is essential that each and every phase in this flow is executed as fast as possible to minimize the total loading time. The developers can take action as soon as the application is loaded and for example present a loading screen. Once the data is fetched the loading screen can be replaced with the real page that the user requested. To keep the file size down it is essential to minimize the time it takes to download and parse the Javascript. Features that are not used by all applications should not be bundled together with the framework, it would simply make the framework bigger than needed. By using a flexible dependency-handling the framework can be configured so that no unnecessary libraries are included.

4.6 Search engine optimization

To approach the problems with crawlers having difficulties running Javascript there are three methods that are widely used:

- Crawler-specific HTML snapshots
- Share templates between front- and backend
- Don't care about crawlers and make no further search engine optimizations

Unfortunately have all of these methods a few drawbacks. If crawler-specific HTML snapshots are generated, then there are two versions of the system to maintain. For every change on the site an implementation is required in both the normal Javscript-based system as well as in the system used by the crawler. This generates a lot of duplication of work, which obviously leads to higher maintenance costs.

The other solution, that is slightly more sophisticated, is to re-use the frontend templates on the backend side. Some template languages, such as Mustache, can be rendered in a wide range of languages. Javascript can be used to render the templates on the frontend, while other languages such as Ruby, Python or PHP can be used to render the templates on the backend [27]. However, even with this solution there exists duplication of work. The router that maps a URL to a given view must now be maintained on both frontend and backend. The API used by the frontend might not be suitable to use directly by the backend, which in that case would requires a bridge layer in between. This layer must now be maintained for every change in the API that is carried through. It also restricts the choice of template language to an alternative that supports both backend and frontend rendering, which is not a very common property of a template language. One could of course argue that if the backend is written in Javascript, like in Node.js, then the same template language can be used both at the front- and backend. Since there are several alternatives for this setup, it would allow a more flexible choice of template language. However, it now restricts the backend to be written in Javascript which certainly is not a suitable choice in all situations.

Not to care about crawlers at all might sound like a bad idea, but it is important to point out not that all websites need to care about search engines. Examples are internal intranets or systems that require authentication, who do not care about whether public users find them or not. That is simply not their purpose.

4.6.1 A new approach

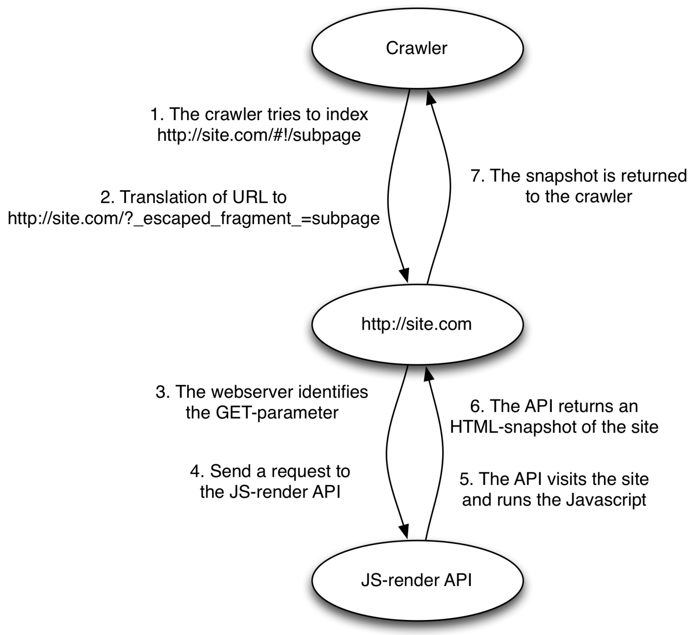
A more flexible solution is to write an API that for a given URL downloads the site, executes the Javascript, and then returns the output. When the crawler visits the site a call is made to the API, which generates a snapshot of the site that can be presented to the crawler. The crawler will not be able to distinguish this from the more traditional solution when HTML-snapshots are generated directly by the backend. The flow of communication is illustrated in figure 4.4 and a more detailed description follows below:

1. The crawler wants to index http://site.com/#!/subpage
2. The crawler recognizes that the URL contains a hashbang and translates it to http://site.com/?_escaped_fragment_=subpage
3. The webserver sees that the visitor is a crawler since the GET parameter *_escaped_fragment_* is set
4. A request to the Javascript-render API is made with the URL http://site.com/#!/subpage as parameter
5. The Javascript-render API visits the given URL and waits for the Javascript to be executed.
6. The Javascript render-API returns a snapshot of the output that is generated
7. The webserver takes the output from the Javscript-render API and provides this to the crawler

This would only require one version of the system to be maintained, that is the normal Javascript-based site. No duplication of work for each view is neither needed, one single solution can be used for the entire site. There are however a number of drawbacks with this solution:

- It heavily depends on the render API. If it crashes it will directly affect the output to the crawler.
- The response time for the crawler's request is delayed. There are now more tasks that have to be completed before the crawler gets the output. Especially the page rendering is critical since it tends to be quite slow. This may affect the search engines' evaluation since slow pages are in general less popular [2].
- It can be hard for the API to determine when the page is ready since content is often loaded asynchronously. A method to determine when the page is completely loaded is needed.

To approach these drawbacks there are a few things that can be done. A cache can be used to store lately generated snapshots in a database. If the cache was recently updated then the page is fetched from there and simply fed to the crawler directly, this would



Crawler
1. The crawler tries to index http://site.com/#!/subpage
2. Translation of URL to http://site.com/?_escaped_fragment_=subpage
7. The snapshot is returned to the crawler
http://site.com
3. The webserver identifies the GET-parameter
4. Send a request to the JS-render API
6. The API returns an HTML-snapshot of the site
5. The API visits the site and runs the Javascript
JS-render API

Figure 4.4: The flow of communication when using an API for HTML snapshot generation.

greatly improve the response time. An own crawler can be written and run periodically to continuously refresh the cache. This would ensure that when the real crawler visits the site the cache is up-to-date and a version of the page can be returned instantly. It also makes the system less dependent on the Javascript render-API since if it happens to crash when rendering a page the last version of it can still be used since it is stored in the cache. This assumes of course that the render-API do not crash *every* time the page is rendered.

To determine when the page has loaded and all the content is in place options can be sent to the Javascript render-API. For example a DOM-selector can be specified so that whenever the DOM-element matching this selector has got some content the page is considered as ready. For more complex pages this might however not be sufficient, then flags can instead be used. Whenever a flag is set to a given value is the page considered as ready, the Javascript render-API can simply poll the flag until the given value appears. This puts the developer in full control of when the page is ready to be served to the crawler.

Chapter 5

Implementation

In order to test how well the architecture would perform an implementation is needed. The framework is implemented in Javascript as described in section 4. To be able to discuss the implementation further on in the thesis a name is needed. The implementation will further on be referred to as MinimaJS.

5.1 Classes and inheritance

Since the architecture is based on modules that all have different properties a class implementation that has a good support for inheritance would be needed. Unfortunately Javascript is using a prototype-based inheritance model with a limited support for inheritance. Even though there is no native support for traditional inheritance the behavior can be emulated, and there are a number of implementations available for this purpose. One of the more popular implementations is written by John Resig, the founder of jQuery. His implementation is based on the models Base2 and Prototype [34]. However, all these models have a behavior that strictly inhibits the traditional inheritance model, but instead have more of a prototype-based approach. Listing 5.1 shows an example of this behavior. As seen in the example c2.obj.counter is set to 1 but the expected behavior in a traditional inheritance model would be 0. This is the case since objects in Javascript are evaluated with a call-by-reference approach while other primitive datatypes such as integers are evaluated with call-by-value.

```
var Parent = Class.extend({
  counter : 0,
  obj : { counter : 0 },
  inca : function() {
    this.counter++;
    this.obj.counter++;
  }
});
var Child = Parent.extend({});

var c1 = new Child();
var c2 = new Child();

c1.inca();

// Prints 1, 1
console.log(c1.counter + ", " + c1.obj.counter);

// Prints 0, 1
console.log(c2.counter + ", " + c2.obj.counter);
```

Listing 5.1: Inheritance using John Resig's inheritance model with shared objects

between instances

In a prototype-based inheritance model a parent instance is shared among all the children. If an object is changed in the parent all children are affected. To get around this each child instance can have its own parent instance. A change in a parent would then only affect one single child, just as expected in a traditional inheritance model. The drawback is of course that it will require more memory to run since there now are more instances stored in the memory. There is no perfect solution to get around this, it is rather a give-and-take situation. If it is important for the design to have a traditional strict inheritance model will it simply require more memory to run, if this is not the case the prototype-based solution should of course be used. Since class inheritance is a crucial part of the system design for MinimaJS, it is based on the solution where each child has its own parent instance.

5.2 Scopes in Javascript

All functions in Javascript are executed on a scope. The scope is accessible through the variable *this*. The scope can be very handy, for example in a class method *this* would typically refer to the current instance of the class. However, due to the nature of Javascript, the code is often written in an event-driven architecture which encourages usage of anonymous functions as callbacks. Whenever these callbacks are fired the scope is changed and *this* is replaced with another object. It is on the other hand possible to control what scope a certain function should be fired on, this can be done with the functions *call* and *apply*. MinimaJS is implemented in a way that tries to maintain the same scope in all possible cases so that the developer does not have to think about its current state. Unfortunately it is in some cases impossible to keep the current scope, a Javascript-developer must always think about when the scope may change.

5.3 API drivers and push synchronization

Two different kind of API drivers were implemented, one for local storage and one for storage on a server. The API driver for local storage on a client is based on the Local Storage API in the HTML5 draft which allows strings to be permanently stored on a client. By serializing data to JSON more complex objects can be stored as well. The other API driver is based on websockets which also comes from the HTML5 draft. Websockets support bidirectional communication between the client and the server [35]. This API driver fully support synchronization between several clients simultaneously.

When several clients are using the system at the same time synchronization problems may appear. These problems appear whenever two clients are doing concurrent transactions against the database. Other more organizational problems may also appear if the users are working with data that is out-of-date, the users might take decisions that is based on the old set of data. These problems can be approached by introducing a protocol for synchronization that is implemented within the API driver. The purpose of the protocol is to allow changes to be distributed among all the clients so that conflicts can be detected and resolved. Each model object is tagged with a revision, when a change is made on the model the revision is increased. The server can detect when a change is made to an old revision and tell this to the client sending the change since this corresponds to a conflict. Whenever a client sends a change request on a model the server will respond with an ack or nack depending on if the change was carried through or not. The API driver must be able to both commit and revert changes on a model. In order to get a good user experience a model is locally changed directly when the user makes a change, first when the model is saved the transaction is sent to the server. The view will therefore be re-rendered directly which results in a more responsive system. When the server answers the transaction will either be committed or reverted depending on if the answer was an ack or nack. In order to get a more simple protocol a synchronous message passing is used, a client is only allowed to send one transaction at a time. Once a client has sent a change to the server must it wait until an ack or nack is received before sending another one. This was implemented using a queue of transactions. An illustration of how the protocol was designed can be seen in figure 5.1. The figure shows when client i has made two changes that are sent to the server. The server accepts the first change and answers with an ack, the seconds change is however out-of-date and is therefore rejected with a nack. All other clients, except for i itself, are notified by the first change that was carried through.

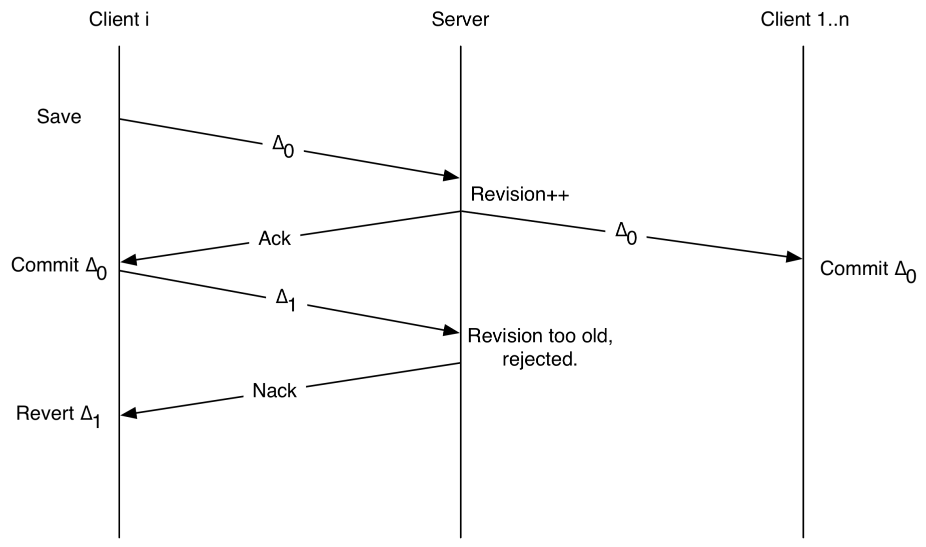


Figure 5.1: An illustration of a client sending two changes to the server.

5.4 Writing testable code

By letting the presentation models be accessible from the module's instance unit tests can be written without interacting with the user interface. Most of the features of the application can then be tested during development with small and simple unit tests. Important to note is that if the unit tests do not interact with the user interface the views would not be tested. On the other hand it will encourage developers to write simple, almost trivial, views that only present the data that its given. A unit test can emulate user input by sending the same event to the controller as the presentation model would do. The result can then be verified by either looking directly at the affected models or listening to events sent by the collection. Every time a view is rendered an event is also sent, by listening to these events the unit test can verify that the correct views are rendered. The unit tests can then validate that the correct data was set and that the right views were rendered. This provides a good basis for a working application.

5.5 Dependencies

MinimaJS has two hard dependencies, one template engine, one selector library. In order to keep the file size low these dependencies are configurable. For example, if jQuery is already used on the site MinimaJS can be configured to use jQuery as its selector library. It would be a waste to include the jQuery library twice. In other cases where jQuery is not used on the site other more lightweight alternatives can be used, such as Sizzle. When it comes to template engines there are several alternatives, each alternative has of course its own advantages and disadvantages. Depending on the application some template languages might be more suitable than others. The template engine can, just as the selector library, be specified in the configuration. At last, the API drivers depend on the JSON library by Douglas Crockford [36] for the serialization/deserialization of the data models.

5.6 Javascript render API for SEO

As explained in section 4.6.1 an API that can render and run the Javascript on a given site is needed. The API is implemented using PhantomJS to render the given page. PhantomJS is a headless webkit browser with an excellent support for Javascript that can be run in a server-side environment. For the API to know when all the asynchronous requests are finished two different triggers can be used, either a DOM-element or a flag can be specified. If a DOM-element is used as trigger the site is considered as completely loaded once this element is updated with some content. The DOM-element is simply polled periodically to check if it has been updated. When instead a frag is used as trigger it is simply polled until a truthy value appears. What type of trigger that is used is set in the Javascript on the site which makes it a quite flexible solution since different pages can use different triggers. If caching is used it will depend on the backend system. In order to test the SEO implementation with cache it was implemented in PHP using MySQL as backend storage.

5.7 Deployment

To minimize the time it takes for a client to download and parse the Javascript a number of actions can be performed. These actions will all together decrease the time it takes before the page can be rendered at the client.

Each time a new TCP connection opens a handshake is performed [28]. The speed is limited by TCP's congestion control which limits the window size. The size of the window is then increased over time until a packet is dropped [29]. The connection is therefore quite slow in the beginning. Since it is likely that an application consists of dozens of Javascript files the total download time will be affected by how new connections are handled in TCP. On top of that are browsers limiting the number of concurrent TCP connections. For example is Google Chrome currently limited to 6 concurrent connections. By concatenating the Javascript files together fewer TCP connections will be opened, and since the window-size grows exponentially over time this will result in a shorter download time in total. Exactly how many files that should be used depends on the application, and tests must be performed to find the optimal number of files for a specific application [30].

To decrease the file size of the Javascript it can be minified using tools such as UglifyJS, YUI compressor or Closure Compiler. A minifier tool tries to optimize the Javascript code to decrease the file size as much as possible. By changing variable names, removing comments and whitespaces the file size can be decreased, but it is of course important that the functionality is kept intact. Other optimizations that can be done are letting the webserver compress all the files before sending them to the client. The gzip compression has a wide browser support which makes it highly suitable to use for this kind of task, for browsers that do not support gzip instead a non-compressed file can be served. By compressing the files with gzip the response can be reduced by about 70% [31]. The gzip format is based on the deflate algorithm which combines LZ77 and Huffman coding. It simply finds common strings and replaces them with pointers (LZ77). Symbols are then replaced with weighted symbols based on their frequency of use (Huffman coding) [32]. This is extra useful for Javascript views that are described in HTML which tend to share long sub-strings. The drawback with compressing the files is that it takes additional CPU cycles to perform which results in a higher load on the web server. On the other hand can the web server cache the compressed files so each file do not have to be compressed at each request.

Another important aspect to consider is DNS lookups. From a lookup perspective it would be ideal if all files were downloaded from the same host that is serving the HTML page. This would minimize the number of lookups since the IP address of the host already is in the DNS cache of the client. If external libraries are used, for example if jQuery is loaded from an external server, CDNs (Content Delivery Networks) should be considered. The idea with a CDN is to distribute data across the internet on servers with high availability and good performance. For example, if a user has visited another site that is using the same CDN for jQuery it is already in the cache of the user and will therefore not be downloaded again. The drawback is that if it is not in the cache of the client another DNS lookup must be performed. The site also becomes dependent on the CDN, if the CDN is not available the entire site will fail to load.

When the frontend becomes big it is sometimes unnecessary to download all the Javascript when the page is loaded since all of the Javascript might not be needed at that specific page. Only the parts that are needed at first are required to be downloaded immediately so that the page can quickly be rendered, the rest of the Javascript can be downloaded asynchronously later on. To know which Javascript files are needed a dependency-tree can be constructed. This describes what other files a given file depends on. For example the page might depend on the framework and the framework itself depends on a number of Javascript libraries, then the page indirectly depends on these libraries. The files can then be divided into and page-specific code and code that is shared between the modules. Tools such as RequireJS can help with the analysis so that the minimum amount of files are fetched synchronously. These tools can be used independently on the implementation, the modules can simply be wrapped into new tool-specific modules which helps the tool to identify the dependencies.

Chapter 6

Verification

6.1 The test application

To test the framework a test application was written. The test application is a todo-list which allows the user to keep track of what needs to be done. Todo-items can be added, edited, marked as completed and removed. Other features such as clearing all completed tasks and marking all tasks as completed were also implemented. The choice of test application and its features was not randomly picked. The open source project TodoMVC has implemented the exact same application in a wide range of Javascript frameworks [33]. The idea behind the project is to make it easier for developers to compare different Javascript frameworks. It also becomes suitable for verifying MinimaJS since it now can be compared with its competitors. The test application can be seen in figure 6.1.

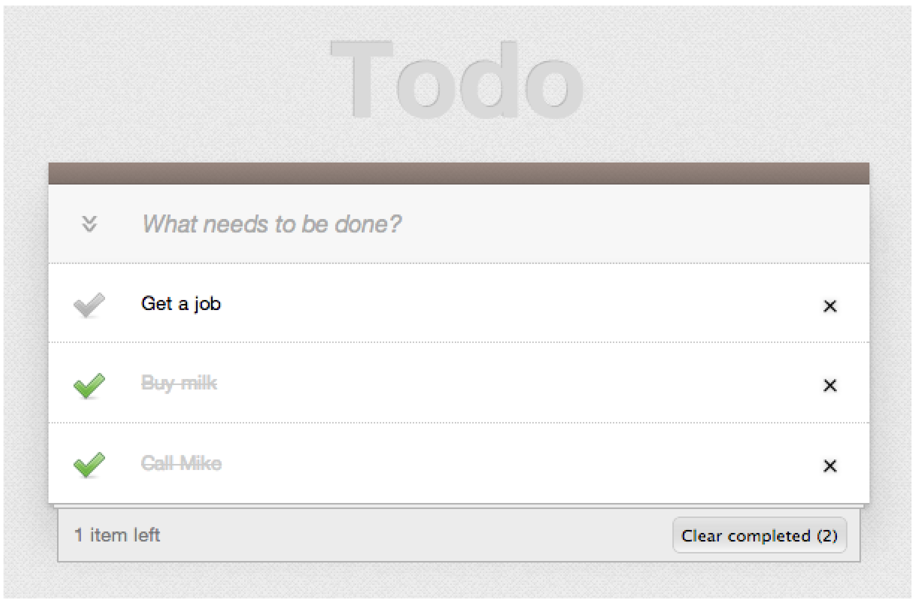


Figure 6.1: The test application.

6.2 Measuring the initial loading time

To see how the framework performed in terms of initial loading time the loading time was measured and compared with implementations in other Javascript frameworks. A comparison was also made against an application that is not depending on Javascript at all, the HTML-files were then simply generated directly by the backend. An implementation written in standard Javascript without any framework at all is also included in the measurements. The files were served through an Apache webserver with the browser running on the same machine as the web server to neglect the transferring time. The transferring time was neglected since focus lies on the performance of the framework and how fast the content can be rendered, not the transferring time itself. The total loading time will always depend on the transferring time but it is of course very dependent on the client. When the measurements were made were there three todo-items in the list, the browser used was Google Chrome 22. Local storage from the HTML5 draft was used as data storage for all the measurements. The result from the measurements can be seen in table 6.1. The result is also illustrated in figure 6.2.

Action	**Static HTML**	**Javascript (no framework)**	**MinimaJS**	**Backbone.js**	**Ember.js**
Download page	48 ms	36 ms	32 ms	30 ms	38 ms
Download scripts	0 ms	37 ms	33 ms	36 ms	51 ms
Parse scripts	0 ms	31 ms	81 ms	85 ms	283 ms
Load application	0 ms	7 ms	7 ms	109 ms	449 ms
Fetch data	0 ms	2 ms	5 ms	14 ms	31 ms
Render page	4 ms	13 ms	17 ms	16 ms	172 ms
Total	**52 ms**	**126 ms**	**175 ms**	**290 ms**	**1024 ms**

Table 6.1: The loading time of the test application in different implementations.

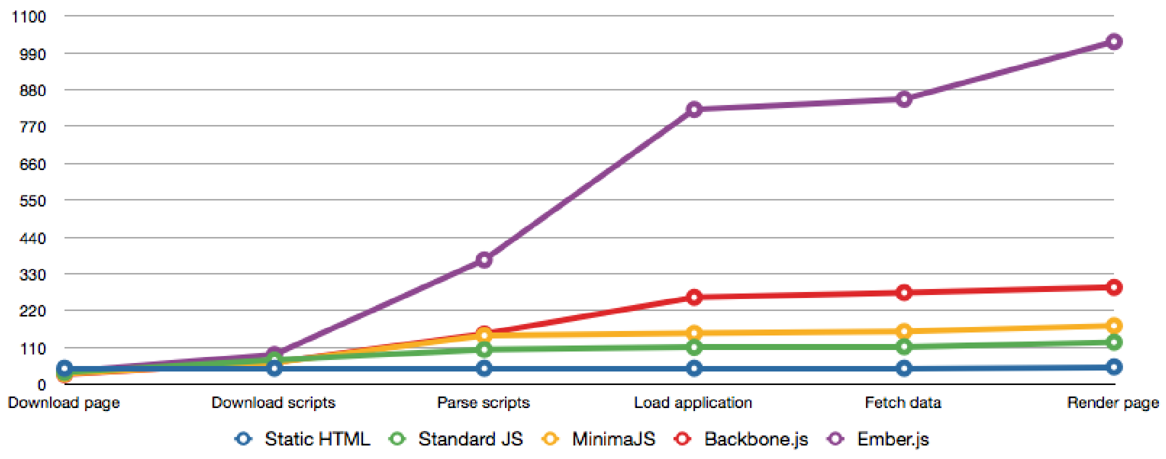


Figure 6.2: A graph showing the loading time of the test application comparing different implementations of it.

As expected the page is downloaded faster on all Javascript-based implementation since it is smaller and does not contain the presentation of the all the todo-items. However, the static HTML does not need to download or parse any Javascript which leads to a shorter initial loading time in total. Considering the low transferring time the time it takes to download and parse of the script is still 65% for the MinimaJS framework. This indicates how important it is with a small file size of the framework. Except keeping the size small it hard is affect the total time until the script has completely been loaded and parsed. Also worth noticing is the poor performance of Ember in this test. Since the Ember-implementation is using two-way data-bindings the result is however not very surprising, one of the main drawbacks with two-way data bindings is the loss of performance. In the end all Javascript versions are slower than the static version which confirms Twitter's criticism [1] against SPAs.

Since the size of the framework essentially affects the initial loading time it can be further analyzed. In table 6.2 is the Javascript frameworks from table 6.1 listed with their respective file size when they have been minified using UglifyJS and compressed with gzip.

Framework	Size of framework	Size of dependencies	Total size
MinimaJS 0.1	4.3 kB	10.4 kB	14.7 kB
Backbone.js 0.9.2	5.6 kB	13.0 kB	18.6 kB
Ember.js 1.0	37.0 kB	0 kB	37.0 kB

Table 6.2: File size of a few Javascript frameworks after minification and compression.

Note that some of the frameworks in the table have a configurable amount of dependencies. The dependencies are chosen so that the frameworks become as small as possible but still support templates and routing. Underscore was used as template language for MinimaJS and Backbone. While MinimaJS could be configured to use Sizzle as selector library Backbone is depending on Zepto which is a bit bigger. Ember.js have no hard dependencies at all. The size of the frameworks matches the initial load time pretty well, a big framework leads to a long initial loading time. As seen MinimaJS is smaller than both Backbone and Ember, on the other hand is MinimaJS in a very early development phase. The size will most probably increase a bit after bugs have been fixed and the browser support has been improved.

6.3 Measuring the testability

To verify the testability a number of unit tests were constructed where each test verified one specific feature in the test application. When these unit tests were executed the code coverage was measured to see how much of the application is used for completing the test. The following features were tested in the test application:

1. Add a new todo-item to the list
2. Read the label of a todo-item from the list
3. Remove a todo-item from the list
4. Toggle one todo-item as completed
5. Set all todo-items as completed
6. Clear all todo-items from the list that are marked as completed
7. Change the label of a todo-item in the list

As described in section 2.6 it is quite messy to write unit tests that make use of the user interface, a better approach is to unit test what is underneath it. The unit tests that were constructed in this test were not allowed to communicate directly with the user interface. When measuring code coverage there are several methods that can be used, in this test was the line coverage measured. The line coverage of an execution represents how many lines of code that were interpreted during the execution, this can be measured in percentage since the total number of lines of code in the application is known. Line coverage not ideal to use in all situations, the number of lines of code does not represent the complexity of the application very well. On the other hand it will give a hint of how much of the application that is being tested. Applications with the exact same functionality were tested for all frameworks. All applications are from the TodoMVC project except the test application for MinimaJS that was implemented in the thesis.

When running the unit tests the tool JSCoverage was used to measure the line coverage, QUnit was used to run and validate the results from the unit tests. The same type of unit tests were used for all of the frameworks that were tested even though they each had to be tweaked in order to work together with the their respective test-application. One test at a time was run, after running a test the line coverage was measured. As a last test all unit tests were run all together so that the total line coverage could be measured. The result from the testing can be seen in table 6.3. Features that could not be tested with a unit test without interacting with the user interface were marked with N/A in the table.

Unit test	Description	Plain jQuery	Backbone.js	MinimaJS
1	Add	N/A	70%	67%
2	Read	N/A	68%	67%
3	Remove	N/A	N/A	69%
4	Toggle	N/A	N/A	70%
5	Set all	N/A	N/A	70%
6	Clear all	62%	73%	74%
7	Change label	N/A	N/A	80%
	All tests at once	62%	75%	90%

Table 6.3: A comparison between different frameworks when measuring the line coverage after a unit test was run.

As seen in the test result is unit testing not a trivial task when writing Javascript applications, but if the system is designed with testing in mind it is possible to get a high degree of testability. To get 90% of line coverage without interacting with the UI at all is not too bad for an application written to serve as a user interface. The line coverage becomes in fact quite high already after one single unit test, and this is the case since there typically is a lot of initialization going on like setting up modules and fetching content from the database. The application written in plain jQuery performs quite poor since its components are tightly coupled and DOM elements are passed around between functions which makes them impossible to test without interface with the UI. The problem with the implementation in Backbone is that some of its functionality is hidden behind a function scope which makes its functions impossible to access. In MinimaJS the presentation models are working as a unified interface that the unit tests can communicate with which make all of its features testable.

6.4 Multiple clients

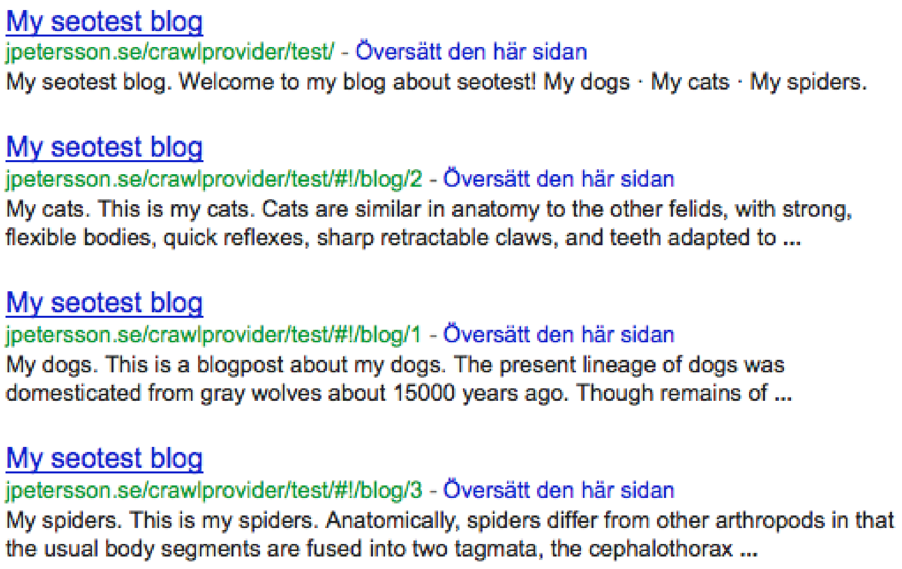
In order to test how well the synchronization protocol worked a stress test was constructed. The idea with the stress test was to see if the set of models in each and every client could be kept synchronized even though a large amount of clients were connected and modifying the data on the same server. The test was using the websocket API driver and the server was implemented in Node.js. Even though the same protocol is used for all drivers there are no reason to test the synchronization aspect for the local storage driver since simply there only can be one single client connected at a time. With websockets on the other hand a large number of clients can interact with the database simultaneously.

To construct a proper test each client would be required to run a separate instance of the framework. However, to run and manage hundreds of GUI windows running the framework is not really a practical solution. Instead webworkers in the HTML5 draft were used. The webworker API allows threads to be spawned in the background running a given Javascript-program, the communication with the threads is done through a thread-safe message passing implementation.

The test program first initialized 100 instances of the framework where each instance was running in a separate thread. When all instances were ready to run a message was sent to them to indicate that they should start their test. The instances then started to do transactions against the database. They randomly picked a valid transaction, either they created a new model or they modified or deleted an existing one. Since all instances were doing their transactions at roughly the same time bursting phenomenons occurred, the server had to handle transactions that lead to conflicts since the same model might have been changed by multiple instances at once. When all instances each had made 10 transactions their version of the data models was sent to the main thread so that they could be compared. If all data from each and every model were the same the test was considered as successful. All instances successfully completed the test and had the same version of their data models.

6.5 Test of a SEO-friendly Javascript site

To test the approach of solving the problem with search engine optimizations for Javascript-based sites a test-site was published. The test-site contained four pages with different content. Once the initial page had been loaded the actual content was asynchronously loaded and rendered by Javascript. If the site was loaded in a browser with Javascript turned off a blank white page was returned without any content at all. The solution described in section 4.6.1 was used to provide the crawler with an HTML-snapshot of the requested page. Once the test-site was submitted to Google the pages could be indexed by Google's crawler. As seen in figure 6.3 did Google's crawler successfully indexed all of the four pages.


My seotest blog
jpetersson.se/crawlprovider/test/ - Översätt den här sidan
My seotest blog. Welcome to my blog about seotest! My dogs · My cats · My spiders.

My seotest blog
jpetersson.se/crawlprovider/test/#!/blog/2 - Översätt den här sidan
My cats. This is my cats. Cats are similar in anatomy to the other felids, with strong, flexible bodies, quick reflexes, sharp retractable claws, and teeth adapted to ...

My seotest blog
jpetersson.se/crawlprovider/test/#!/blog/1 - Översätt den här sidan
My dogs. This is a blogpost about my dogs. The present lineage of dogs was domesticated from gray wolves about 15000 years ago. Though remains of ...

My seotest blog
jpetersson.se/crawlprovider/test/#!/blog/3 - Översätt den här sidan
My spiders. This is my spiders. Anatomically, spiders differ from other arthropods in that the usual body segments are fused into two tagmata, the cephalothorax ...

Figure 6.3: The result after submitting the test-site to Google's Crawler.

Chapter 7

Discussion

7.1 Test-driven development in Javascript

Test-driven development is something that is not used very much today by Javascript developers. If a framework can provide an architecture that helps developers to get around the problem with writing simple unit tests it will most probably be used more than it is today. As seen in the thesis it is possible to achieve high degree of testability if just the testing aspect of the application is considered already in the design phase of the system.

7.2 The importance of delays

The initial loading time has been considered as an important aspect of SPAs in the thesis since it directly affects the users. If the users are not satisfied the website will become less popular. However, another study suggests that the time it takes to interact with the website is even more important than the initial loading time [37]. Users were asked to rate the quality of a service when the delay was varying. The result showed that the tolerance for delay is decreasing over time. Users were more patient the first time the page loaded than when interacting with it later on. This explains why SPAs are so popular, they have a slow initial loading time but that is also when the users are as most patient. It also highlights why SPAs are well suited for websites with a high degree of user interactivity. SPAs perform well after the page has loaded which corresponds to when the patience is decreased.

7.3 When to use SPAs

As explained above SPAs are great for all websites that involve a high degree of user interactivity, like websites that are more like applications rather than traditional web pages. They are also great to use for websites that target mobile devices. In some cases SPAs are even considered as an alternative to native mobile applications. If a mobile application is written as a web application there is only one version to maintain. If instead a native application would be developed there would be one version for each operating system. It would of course result in both higher development and maintenance costs. On the other hand the user experience will be a lot better in a native application since it always will be able to perform better than a web application.

For websites that are dependent on very precise SEO SPAs are most probably not such a good choice. Unless the crawlers start to execute Javascript will a separate solution for SEO always be needed. All these solutions have their limitations, a traditional website does not. Today it is often hard to get tools for web analytics to coop with SPAs, these tools are essential to develop a good SEO strategy. The problem is that since a SPA never reloads the page the user actions will not be posted to the analytics tool, which results in the visitor flow not being analyzed. Another problem with SPAs is that users with accessibility tools, such as screen readers, might have a hard time using the application. Most of these tools have a limited support for websites that asynchronously changes its content, which of course will make SPAs hard to use.

7.4 Future work

An interesting question to ask if whether MinimaJS is a realistic alternative to existing frameworks on the market. The framework is of course in an early development phase and more testing is needed before using it in production. Especially browser compatibility is an important aspect when targeting a wider market, this is an aspect that has not been considered at all during the development. However, there is nothing in MinimaJS that is impossible to implement to support the browsers that are popular today. To get it to work in all major browsers it would probably not require too much work. Documentation is another thing that has to be improved, it is an essential aspect to consider to enable other developers to learn the framework.

Since the framework performed very well in the tests compared to its competitors the future is looking bright. Especially when it comes to testing which is something that is becoming an even more important part of the development.

Bibliography

[1] Dan Webb, Twitter, 2012-05-29, *Twitter engineering blog*
http://engineering.twitter.com/2012/05/improving-performance-on-twittercom.html
[2012-06-11]

[2] Sean Work, 2011, *How loading time affects your bottom line*
http://blog.kissmetrics.com/loading-time/ [2012-07-06]

[3] Nicholas C. Zakas, Yahoo, 2010-10-13, *How many users have JavaScript disabled?*
http://developer.yahoo.com/blogs/ydn/posts/2010/10/how-many-users-have-javascript-disabled [2012-08-15]

[4] Ali Mesbah, Arie van Deursen, Delft University of Technology, 2006, *Migrating Multi-page Web Applications to Single-page AJAX Interfaces, 2nd revision*, Software Engineering Research Group, Delft University of Technology

[5] Valerio De Lucaa, Italo EpicocoDaniele Lezzi, Giovanni Aloisio, 2012, *GRB_WAPI, a RESTful Framework for Grid Portals*, p. 2, DOI: 10.1016/j.procs.2012.04.049

[6] Google, 2012-02-17, *Making AJAX Applications Crawlable - Learn more*, What the user sees, what the crawler sees
https://developers.google.com/webmasters/ajax-crawling/docs/learn-more [2012-06-28]

[7] Tom Dale, Yehuda Katz, 2011-03-01, *Sproutcore, Getting started*
http://guides.sproutcore.com/getting_started.html [2012-06-25]

[8] Angular JS, *Angular JS - What is a Module?*
http://docs.angularjs.org/guide/module [2012-06-25]

[9] Ron Jeffries, Grigori Melnik, 2007, *TDD: The art of fearless programming*, p. 29

[10] Roy W. Miller, Christopher T. Collins, 2001 *Acceptance testing*, p. 1, XP Universe

[11] Addy Osmani, 2011, *Writing Modular JavaScript With AMD, CommonJS & ES Harmony*
http://addyosmani.com/writing-modular-js/ [2012-06-25]

[12] Addy Osmani, 2012, *Learning JavaScript Design Patterns*, O'Reilly media, Inc.

[13] Ross Harmes, Dustin Diaz, 2008, *Pro Javascript Design Patterns*, p. 12, Springer-verlag New York, Inc.

[14] W3C, 2012-03-29, *HTML5 - A vocabulary and associated APIs for HTML and XHTML*
http://www.w3.org/TR/2012/WD-html5-20120329/history.html#history [2012-06-25]

[15] Matthew MacDonald, 2011, *HTML5 - The missing manual*, p. 373, O'Reilly media, Inc.

[16] David Fanagan, 2011, *Javascript - The definitive guide, 6th edition*, O'Reilly media, Inc.

[17] Stone Tapes, 2012-05-25, *Google now searches Javacript*
http://www.i-programmer.info/news/81-web-general/4248-google-now-searches-javascript.html [2012-07-10]

[18] Google, 2012-02-17, *Making AJAX Applications Crawlable - Full specification*
https://developers.google.com/webmasters/ajax-crawling/docs/specification [2012-06-28]

[19] T. Berners-Lee, 1994, *RFC 1630*, p. 13
http://www.ietf.org/rfc/rfc1630.txt [2012-06-28]

[20] R. Fielding, UC Irvine, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, 1999, *RFC 2616*, p. 31
http://www.ietf.org/rfc/rfc2616.txt [2012-06-28]

[21] Angular JS, *Angular JS - Data Binding in Angular*
http://docs.angularjs.org/guide/dev_guide.templates.databinding [2012-07-02]

[22] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1994, *Design Patterns - Elements of Reusable Object-oriented Software*, Addison-Wesley.

[23] Eric Freeman, Elisabeth Freeman, 2004, *Head first - Design patterns*, p. 530, O'Reilly media, Inc.

[24] John Smith, 2009, *WPF Apps With The Model-View-ViewModel Design Pattern*
http://msdn.microsoft.com/en-us/magazine/dd419663.aspx [2012-07-05]

[25] Addy Osmani, 2012, *Understanding MVVM - A guide for Javascript developers*
http://addyosmani.com/blog/understanding-mvvm-a-guide-for-javascript-developers/ [2012-08-30]

[26] Davy Brion, 2010-07-21, *The MVVM Pattern is highly overrated*
http://davybrion.com/blog/2010/07/the-mvvm-pattern-is-highly-overrated/ [2012-07-05]

[27] Mustache, *Mustache - Logic-less templates*
http://mustache.github.com/ [2012-07-06]

[28] Behrouz A. Forouzan, 2010, *TCP/IP Protocol Suite, fourth edition*, Connection establishment, p. 442, McGraw-Hill Companies, Inc.

[29] Behrouz A. Forouzan, 2010, *TCP/IP Protocol Suite, fourth edition*, Congestion control, p. 473, McGraw-Hill Companies, Inc.

[30] Nicholas Zakas, 2012, *Maintainable Javascript*, p. 145, O'Reilly media, Inc.

[31] Steve Souders, 2007, *High Performance Web Sites: Essential Knowledge for Front-End Engineers*, p. 31, O'Reilly media, Inc.

[32] L. Peter Deutsch, 1996, *RFC 1951*, p. 13
http://www.ietf.org/rfc/rfc1951.txt [2012-08-30]

[33] Addy Osmani, Sindre Sorhus *TodoMVC*
http://addyosmani.github.com/todomvc/ [2012-08-29]

[34] John Resig, 2008-03-20, *Simple JavaScript Inheritance*
http://ejohn.org/blog/simple-javascript-inheritance/ [2012-07-30]

[35] Ian Hickson, W3, 2012-08-09, *The Websocket API*
http://www.w3.org/TR/2012/WD-websockets-20120809 [2012-08-15]

[36] Douglas Crockford, 2010-11-18, *JSON in Javascript*
https://github.com/douglascrockford/JSON-js/ [2012-08-17]

[37] Nina Bhatti, Anna Bouch, Allan Kuchinsky, 2000, *Integrating user-perceived quality into Web server design*, Elsevier Science B.V.