UNIVERSITY OF OSLO
Department of Informatics

Investigating Modern Web Applications

From Thin Clients and SQL to Thick Clients and noSQL

Master thesis

Michael K. Gunnulfsen

Automn 2012

Contents

Part I

Introduction

Chapter 1

The Thesis at a Glance

Web 2.0 is a popular term for the second generation World Wide Web. A new paradigm has emerged with the Internet's changing usage pattern that is increasingly becoming more social [**web20**]. In typical Web 2.0 sites, the users and their activities is the main information content, and users are actively involved by connecting and interacting with each other. Examples of such web sites are Internet blogs, chat services, wiki's, video sharing sites, mashups and other social networking services. A characteristic property with such an application is that there are a lot of simultaneous users all in which communicates with each other asynchronously on the Internet. Users registers themselves on the social networking site by entering information that fits their personal profile. After registering, the users can normally search for other users and send requests to connect. These connections makes up a graph of interconnected users, hence the term "social network" (REF) . Based on the information the users stores about themselves they can get various personal recommendations like what movies a given user probably likes, suggested friend-connections, suitable dining places etc.

A characteristic property with Web 2.0 applications is that the information-content on the web pages are primarily generated by the users. Having the users generate and share content with each other often leads to huge amounts of data stored in the application's databases. This differs from other web sites where the creators (or owners) themselves generates the content on the pages, and the amount of data is seldom of such large quantities. In addition to this, another characteristic property with Web 2.0 applications is that the sites often offers highly rich and interactive user-interfaces. The web pages contains graphical widgets that displays animated behavior, and clickable components that generates interactive responses. Of this particular reason, such interactive web sites are often referred to as web applications, or in short "web-apps" rather then web pages (referanse!).

A typical Web 2.0 application is a system that brings a challenging requirement for it's software architecture. Common usage behavior normally requires many and frequent data retrievals and updates. Considering the fact that the amount of data that is stored in a typical web-app is often very

large, it is obvious that the backend architectures needs to be both scalable and efficient. The high amount of dynamic and interactive user interfaces requires much, and often complex graphical user interface code. This code has to be flexible and maintainable, in order to facilitate new user requirements, something that happens to occur especially frequently for popular Web 2.0 sites. (REFERENASE) Also, the user interface code required for interactive web-apps has to be at most efficient, in order to achieve proper responsive behavior. In addition, the amount of users that simultaneously accesses the web-apps are often high. This puts a high demand for scalability on the server(s) that hosts the given application. To meet with these requirements, application developers has to pursue a software architecture that both performs fast and is scalable.

It is not without reason that the quality of the user interfaces of modern web applications has increased dramatically the last years. Rich user-experiences primarily relies on efficient web browsers, because it is the browsers that executes most of the dynamic behavior of a web-app. Earlier, demanding front-end behavior had to be implemented by technologies like Flash and Java applets, mainly because these technologies executes in efficient run-time environments. Older JavaScript engines were not that efficient, so the purposes of JavaScript were mostly limited to simple graphical behavior. However, modern JavaScript engines have become so powerful, that the browsers are now able to execute complex and intensive JavaScript code. This has facilitated the design of rich and interactive user-experiences that can be seen in modern web-apps.

The traditional software architectures for web applications has in the last decade followed a thin-client/thick-server approach. These architectures consists of a client tier, a logic tier and a data tier, where most of the application's source code is executed on the server (backend), and the data tier is often implemented with a relational database. The logic tier is responsible for creating and delivering a dynamically generated HTML page to the client, upon a URL request. The HTML code delivered to the client contains just about enough JavaScript functionality to generate the necessary interactive behavior. However, the rise of Web 2.0 has brought some interesting technologies that has made it possible to implement full applications primarily in the client tier (i.e the HTML, and JavaScript that executes in the browser). This has for long been pointless, because some five years ago, browsers weren't even able to host large-scale JavaScript applications. Also, large-scale JavaScript web applications are difficult to implement and maintain, because the language itself lacks idioms and semantics that suits substantial codebases. However, modern JavaScript frameworks provides syntactic sugaring enhancements that improves the programming experience, which makes it easier to build front end applications in pure JavaScript. All together, these modern technologies might simplify the development of interactive and scalable web applications.

Lately, one has seen a large amount of innovating and pioneering changes in the way developers tend to design modern web architectures. The classical three-tier architecture with a fat server performing all of the

application's business logic and old SQL database is no longer necessarily the best solution for every web application. Even though, there are still many advantages with the traditional approach, considering the architecture has been a de-facto standard for many years. The work that has been done in this thesis is an analysis of the pros and cons for developing a traditional Web 2.0 application by using either a traditional web application architecture, or a modern and experimental web application architecture that is empowered by modern web- and database technologies.

1.1 The Hypothesis / Problem Statement

In this section, we look at the main questions we want to solve in the thesis. We use the term "Web app" to refer to typical modern Web applications that contains rich and responsive user interfaces, incorporates social networking, and manipulates large amounts of persisted data. The main question we ask concerns the benefits, if any, for having a **thick client**/thin server Web app architecture instead of a **thin client**/thick server architecture. We separate this problem statement into three more specific questions:

1. In traditional Web apps, dynamic HTML pages are generated on the server, which is done in every Url request. Can Web apps achieve better performance and scalability results by rendering pages on the client, by using the server only as an interface to the database?

2. In traditional Web apps, state is kept on the server. Can Web apps perform and scale better by moving state completely to the client? Also, if state is moved to the client, can the Web app benefit from moving the business logic operations to the client as well? Benefits are evaluated in terms of programmer satisfactory aspects like code flexibility and maintainability, and also performance.

3. In traditional Web apps, the data is persisted in a relational database management system by using SQL as the query language. However, new types of database systems (commonly called noSql) offers a different, schema-less persistency solution that is often specialized to fit a specific type of application's data modeling need. Is there any such database system that can make Web apps perform better then a typical SQL database implementation? Are there any other advantages of using such a noSql database in Web apps, like programmer satisfaction?

1.2 Goals

The goal for this thesis is to find a superior software architecture for the type of traditional Web 2.0 application that was defined in the beginning of this chapter. Modern Web applications are commonly built using traditional thin client architectures, but thick client solutions are getting

increasingly more popular. Therefore, we will investigate modern and innovative Web architectures in order to find the pros and cons in choosing a traditional Web architecture, versus a modern Web architecture.

Obviously there will never be one architectural design that is the best fitting design for every web 2.0 application. After all, Web 2.0 is just a common name for web applications that are meant for social user interactions, and these applications varies in many ways. Therefore, the goal for this thesis is not to find a solution that is best suited for one particular application. Instead, the objective is to engage the common principles of such applications, like rich user interfaces and domain data made up of interconnected users, blog posts, comments and tags. These principles will be applied in the development of a Web 2.0 application that is to be deployed on a test server. Hence the goal is to find architectural **design principles** that are well suited to solve the most common properties of a web 2.0 application.

The experiments that has been performed on the prototype serves to investigate how modern web technologies can be applied to achieve a scalable and responsive web application that delivers dynamic and interactive content to a big amount of simultaneous users. The goals are to find the bottlenecks and benefits in a classical thick-server architecture, using a traditional SQL database. At the same time, I have studied the benefits and pitfalls from applying a thick-client JavaScript based web architecture, with a simple backend that runs a noSQL database.

In a classic web 2.0 application it is at most important to have a code base that makes it easy to add new features. Therefore, part of the goal of this project is to design a software architecture that is flexible and maintainable, so that one can easily add new behavior and modify existing features. It is a known fact that JavaScript code tend to be tightly coupled with the HTML and CSS styling it dwells in, and therefore, it is an important design criteria that the thick client architecture is both well-structured and code-intuitive. I will strive to apply appropriate design patterns both on the backend and especially on the client, to enforce a flexible and maintainable code base.

It is important to have in mind that a server backend can only be scalable up to a certain point, in which case the only solution to achieve further scalability is to upgrade the server hardware, or add more physical servers. Considering that this is both resource demanding and time consuming, I will limit my conclusions to apply only for single server deployments. As for the client users, the goal for the prototype is to work ideally for modern web browsers, as older browsers lack performance ability to execute large-scale JavaScript applications. Therefore the prototype will only be tested on modern web browsers.

Even though security is also a big issue when it comes to designing Web applications, I have decided not to focus on it, simply because it would be too time consuming and laborious. The only security issue I bring to focus is authentication, because it has a very high impact on the overall architecture of Web applications. The goal is not to find the best authentication protocol for any Web-app architecture, but rather to find

simple solutions that fits the architectures we will discuss.

1.3 Approach

Considering the main goals for this thesis is to compare two different approaches to web application architecture, a good way to get good and reliable results is to design and implement a software application, and use this as a base for experiments. The goal is to identify the pros and cons with a traditional web architecture, versus a modern and experimental architecture. For this thesis I have come up with an idea for a traditional web 2.0 application that conforms to the user requirements stated above. The application is a social networking site where users connect to each other, and creates and posts blogging content. Users will perform common blog related actions like commenting, tagging and give ratings. The application is built twice from the start with two completely different architectures. The first is a traditional thick-server/thin-client model that uses a SQL database to store its data. The server encapsulates all the business logic, which is implemented in a commonly used programming language. The other is a thick-client JavaScript based web application that just uses a backend server as means to store persistent data. This means that most of the business logic is located in the client tier. It uses a noSQL database to store the data.

To get good and valuable results, extensive system-testing has been done on both prototypes. There has been proposed a set of concrete test cases that has been executed once for each of the two architectures. The test cases are designed to test the performance and scalability behavior of the applications. To be able to do this, a lot of dummy data has been generated and used to populate the databases. In addition, the testing was done by running test simulations that generates a high number of simultaneous requests. For most of the test cases the number of requests was increased until the server could no longer scale to deliver reasonable responses.

1.4 Proposed solution

For this thesis I have come up with a web application that I call *Shredhub*. This is an approach to a classical web 2.0 application, primarily designed for musicians. The web-app allows users to register themselves as musicians, deploy videos of them playing and tag the videos with various categories. Other users can watch the videos, rate them and leave a comment. Users connect to each other in a typical social-networking manner, and they receive suggestions for other users they might be interested in connecting to. The web app contains some graphical widgets that requires dynamic JavaScript behavior, and the amount of data that is persisted grows exponentially with the amount of interactive users.

For the purposes of this thesis, the application is built twice with two completely different software architectures, where the first conforms to a traditional web architecture, and the latter incorporates principles

of modern JavaScript-based web architectures. The two approaches are respectively named *Architecture 1.0* and *Architecture 2.0*.

Architecture 1.0 is a built with en emphasis on a thick server implementation that contains all the business logic needed for the application to work as intended. The server is built in Java, which is a commonly used programming language for backend web applications. The server works by responding to URLs that is initiated by the users. For each URL request the server would build and render an HTML page that contains the necessary JavaScript functions to deliver dynamic behavior for the given page. The HTML page is sent back to the user's browser, which does not have to perform much JavaScript processing, since the HTML page is already pre-rendered and ready to be painted on the screen. The whole backend service is built around the MVC design pattern, and incorporates other architectural design patterns for structuring the codebase.

As for the persistency layer, the solution uses a PostgreSQL database, which is a popular open-source implementation of a relational database management system. The backend server that is written in Java contains code to communicate with the database. The front end tier is implemented in JSP which is a templating language that is transferred into HTML during each user request. The HTML code that is generated contains self-contained and independent JavaScript functions that is mixed inside the HTML code. Also, client state is being kept on the server, meaning that the server has state information for every currently logged-in user, stored in memory. Hence the server depends on a session identifier in each URL request in order to identify the user's session. This identifier is stored in a *cookie*.

Architecture 2.0 is built with an emphasis on modern web technologies and business logic running mostly in the client tier. The backend tier is composed of a simple web interface that receives initial requests for the web site, and responds with a big assembly of JavaScript files that makes up the web application. After the initial request, the backend server answers to data requests that targets the persisted data objects stored in the database. These objects are returned from the server in a fine-grained transfer format. This client-server interaction is very different from that in *Architecture 1.0* where the server always reacts by rendering and returning complete HTML pages. The client tier is structured around a variation of the MVC pattern, and implemented with a JavaScript framework called BackboneJs. This framework helps structuring large-scale JavaScript applications by integrating modules, utility functions and event handling in the JavaScript tier.

The JavaScript tier is responsible for constantly rendering HTML pages based on the user's actions. From the programmers perspective, these HTML files are completely separated from the JavaScript files, and the JavaScript files are separated in loosely coupled modules. By using a technique called asynchronous module loading, it is possible

for every JavaScript module to define which HTML pages and other JavaScript modules it depends on, and further load these dependencies asynchronously each time a given JavaScript module is executed. This enhances the interactive experience from a user's perspective, because the browser never blocks while loading in new modules.

Also, the client state is stored in the client's browser, so that no state information is kept on the server. This is done by using HTML5 local storage, which is a simple key-value container that is persisted in the browser. This way, user events are handled primarily by the JavaScript tier, and the JavaScript tier will only contact the server when it needs to access persistent data. The persistency data itself is stored in a noSQL database called MongoDB, which is an open source, document-oriented database technology. It's data is stored as JavaScript objects, which makes a consistent interaction model across the whole software stack in the web application.

1.5 Evaluation

The evaluation is purely based on tests that has been performed on the Architecture 1.0, and Architecture 2.0. The tests were designed to track down efficiency, scalability and to some extend source code maintainability. A testing framework has been used to create fictive test stubs that imitates browser requests. This framework has the ability to create hundreds of thousands of simultaneous running threads, such that the architectures can properly be evaluated in terms of how well they respond to these tests. In addition, a set of tools to measure the client-server roundtrip time, and to track the browser's rendering process upon page loads, has been used to test and the efficiency of the two architectures.

Efficiency has been evaluated in terms of the response time (in milliseconds) when an action is performed in the web app. The action might be clicking a link in a tab that leads to a new page, uploading a video, logging into the web app etc. Considering that the response time is not just a single result when it comes to performing an action, because the process of rendering a web page is a matter of multiple request-response cycles, the evaluation is based on multiple factors. This includes how fast the browser displays an initial result, or in other words, how long the user has to wait before he sees anything at all, how much time it takes before the user can start interacting with the page, and how much time it takes before the whole action is complete.

Scalability is evaluated in terms of how well the web app deals with an increasing number of simultaneous requests. This has been done by creating multiple threads that simultaneously executes the same action on the web app. The evaluation is then based on, for each number of simultaneous request in the set of *Sq={1,10,100,500,1000,10000,100000,500000}* where Su is simultaneous requests, how fast is results being delivered, and how

many requests can the app at most handle before it no longer returns valid answers.

Source code maintainability is only a minor evaluation point in this thesis (much due to the limited amount of time there is to test this in this thesis). However, considering that this is also very relevant in terms of comparing the two software architectures, some evaluation has been done. The two codebases are compared in terms of the amount of lines of source code, the number of different programming languages used, and lastly, how much code had to be modified and added when a new user requirement was introduced and was to be implemented in the already finished codebase. The latter test case was designed to involve both the implementation of a graphical user interface component, a business logic operation, and a new database operation.

All tests has been done when the two web apps were deployed on a web server stationed in Stockholm, Sweden, in order to get reasonable transmission delays. Also, in order to test the performance results on a relevant set of browsers, the efficiency tests has been tested on both the newest version of Google Chrome, Mozilla firefox and Internet Explorer.

1.6 Problem statement

Current Internet devices like desktop computers, smart phones and tablet devices has a high capability for performing resource demanding processing jobs and intensive network transmissions. This makes it possible to build performance demanding front-end applications that can run on these devices. However, the traditional three-layered web architecture is often built with an emphasis on limiting the amount of processing that is to be executed on the client's device. This is done by leveraging the high processing power of the web servers, such that most of the demanding computing tasks are performed on the servers, and the finished HTML-results are prepared and sent to the front-end clients. It is interesting to study how web applications can benefit from moving much of the business logic into the client's device instead of letting the server perform most of this computation. The fact that a classical Web 2.0 application requires lots of simultaneous users with high amounts of data requests, makes it interesting to see if such an architecture might result in less load on the server. However, having lots of business logic on the client means lots of JavaScript code, since JavaScript is the one and only cross-platform language supported by all web browser(REF?). Now, it so happens that the JavaScript programming language has ever since its dawn of time been used just as a supplement to HTML pages, just to provide the necessary dynamic behavior. The language itself is not intended to develop large-scale software applications, and it is therefore challenging to build a big and maintainable JavaScript code base. To cope with this, web developers has spent lots of time lately to build open-source JavaScript frameworks that serves to ease the challenge of

creating large-scale JavaScript web apps. This makes it interesting to study the benefits of developing a modern Web 2.0 application with a pure JavaScript implementation, rather then a backend-oriented traditional software architecture.

Another interesting scenario with Web 2.0 apps is the massive amount of data that is being persisted in these kinds of applications. It is the users that creates the application's content through blogs, wikies and social interactions, and the data magnitude expands with the increase of users and their interconnections. With the demand to persist such large data quantities, and to perform efficient operations on them, one has seen lots of new and innovate database technologies. These new types of database systems all have in common that they distinguishes themselves from the traditional relational database. The aims for these generally focuses on the ability scale over multiple machines (horizontal scaling), and perform highly efficient on large persistent data quantities. This paradigm has been named noSQL, and examples include document-oriented databases, column-oriented databases, and key-value databases. A common factor with these noSQL databases is that they have showed to perform great in cloud environments because of their horizontal scaling abilities (unlike traditional SQL databases). Naturally this fits great with the requirement of a traditional web 2.0 application, because it requires high performance in data lookups even when the size of the data quantity is extremely large. Still however, many developers favors thetraditional SQL database implementations because of its elegant schema design, efficiency, and also the fact that most developers has profound knowledge and experience in the field of relational database management. The question of wether a noSQL database has any benefits compared to using a traditional database in a thick vs thin client architecture is an interesting topic. Besides the scalability question, which seems fairly self-proven, there are other not so revealing problem statements, like what sort of queries are best suited for the various database technologies.

The problem statement in this thesis is centered around two main topics:

1. Can a thick client web architecture help make an interactive web 2.0 applications perform better? This must be evaluated in terms of performance, in which the response time for end-users is the main evaluation criteria. This must be tested both when the number of simultaneous requests are small, and when the number increases. This must be done in order to simulate a real-world scenario of a modern web-app. Also, scalability must be evaluated in terms of the ability for the backend server to deliver reasonable responses when the number of simultaneous requests increases. The problem statement is limited to a single server environment, simply because proper testing in a multi-server environment is too time-consuming for this thesis. An additional problem statement is the question of wether there are any programming benefits from building a large-scale JavaScript application, compared to developing a traditional thick server architecture

built in a traditional programming language like Java. Just like in the traditional approach, the large-scale JavaScript application is built around the MVC design pattern, in order to achieve a good structuring of the codebase. This makes it feasible to compare the quality of the two source codebases. The programming benefits must be evaluated in terms of intelligible, and structure. However, the task of evaluating flexibility and long-term maintainability is very demanding and time-consuming. Therefore, the problem statement is limited to simple comparisons and evaluation of the structure and intelligibility of the two codebases.

2. Are there any noSQL databases that are particularly more suitable for a typical web 2.0 application than ordinary SQL databases? The problem statement is especially relevant for web 2.0 applications, because of the type and amount of information that is stored in such applications are extraordinary compared to other types of web applications. The persistence technologies must be evaluated in terms of efficiency and scalability. Is it the case that it's always better to use one of the other, or are there any special types of operations or queries that are better solved by any of the particular database technologies. Like with the problem statement above, programming satisfaction must also be a design goal. In this case, simple evaluation criteria like ease of development and intelligibility will be considered, in order to maintain an appropriate scope for the thesis work. Now, it is so that there are a lot of various noSQL, and SQL databases, so the task of finding the best solution for all of these is simply to overwhelming for this thesis. Therefore, the problem statement is limited to finding one representative noSQL database, one and SQL database, and the evaluation will be based on a performance measurement on these. The database technologies chosen must be widely used in the web application industry, which is important in order to get valuable results.

1.7 Work done

The initial work done for this thesis was to identify common characteristics in web 2.0 applications. This lead to the design of a web application that could incorporate these characteristics in the application's user requirements. The application was first designed from a user's perspective with an emphasis on rich user interfaces, with dynamic and interactive behavior. At the same time, a lot of work has been done in studying architectural trends in modern web applications. The author of this thesis has spent much time looking at open-source code repositories, read technology blogs, books, watch web-seminars and presentations, and read online discussions on web architecture. It so happens that not a lot of research has been done on modern web application design, so much of the knowledge is based on the sources just described. It was important to get a comprehensive overview of the common trends in web architecture

in order to decide the most suitable solutions for the prototypes that was developed in this project.

Further, the work involved the design and implementation of the two prototypes. The implementation process had a strong focus on keeping the applications look exactly the same from a user's perspective, while at the same time focusing on developing the architecture in two completely different ways. A list of the code that has been implemented in given in the list below. Note that the list applies to both of the prototypes:

- Design and implement user interfaces with HTML, CSS and JavaScript
- Implement business logic, including validation rules and business processes
- Implement database scripts to generate the databases
- Implement code to communicate with the database
- Implement backend functionality to communicate with client users through HTTP
- Create scripts to generate dummy data
- Deploy the prototypes on a self-hosted server
- Test the implementation to verify that it works as expected

The last part involved deciding how the two architectures was to be tested and compared. This work involved defining a set of concrete test cases aimed to testing the performance and scalability behavior for the applications. The test cases are designed to identify efficiency properties, bottlenecks and capacity limits. Finally the tests were run on the test machine that was hosting the applications. The tests were performed both physically by an active user, issuing requests from the browser and registering the round-trip times, and the were performed by using testing tools that are are able to create multiple dummy requests. The testing tools where used to check the scalability behavior be increasing the number of simultaneous requests, until the server were no longer able to respond with reasonable results, or crashed.

In addition, the two codebases where revisited and compared in terms of flexibility and comprehensibility. The work was a rather short evaluation including a structured table-comparison of the significant pros and cons. This was important to get a complete overall evaluation of the two software architectures.

1.8 Results

The results shows that the modern architecture achieves better performance results both for responds time, and backend scalability. The reason is

that since it avoids involving the server as much as possible, less data is sent between client and the server. Also, it achieves an interactive user experience by following an asynchronous page loading approach, giving the user the impression that the page instantly responds to requests. In addition, the noSQL implementation achieves very high quering speed, because it avoids doing tedious join operations between multiple tables. This results in higher scalability for Archtiecture 2.0. A benefit with Architecture 1.0 however, is that the solution is easy to implement for most developers, because the coding style is very familiar. The result is that many programmers will probably spend less time developing the first architecture then the latter.

1.9 Contributions

1.10 Criteria

In chapter 1 : security is at most a secondary concern. Its only authentication.

Chapter 2

Background

2.1 Introduction

Web applications has in the last few years seen a dramatic change in both behavior and magnitude. They have grown from being a set simple consistent web pages into highly dynamic, and interactive applications with rich user interfaces. Previously, interactive behavior in web sites were usually performed by Java applets and Flash applications [56] that could run inside the browser. But as JavaScript engines and web browsers has become significantly powerful, such behavior is increasingly being implemented exclusively with JavaScript.[56] Together with this shift towards highly interactive web applications, the user behavior is at the same time increasingly becoming more social. Users makes up the main data content of web applications by socially interacting with each other and adding content to the pages.

This chapter outlines recent trends in applications that can be found on the Internet; commonly named as Web 2.0.[60] We will look at the technologies that enables applications to run on the Internet, and more specifically, the software architectures and technologies that are commonly used for developing traditional Web 2.0 applications.

The chapter begins with a short history of the world wide web, then a discussion of how the web has changed from being simple and static web documents, into dynamic Web 2.0 applications. Then, we present an overview of the key attributes and common user behavior that is found in modern web applications. Finally an overview of the software architectures and technologies that are commonly used to implement such applications will be given. This background material will be the foundation of the study that has been done in this thesis.

2.2 From Web Sites to Web Apps

2.2.1 History of The World Wide Web

The World Wide Web (www) was first introduced by Sir Tim Berners Lee at the CERN research laboratory in 1989 [46]. He laid out a proposal for

a way of managing information on the Internet through hypertext, which is the familiar point-and-click navigation system to browse web pages by following links. At this time, Tim Berners Lee had developed all the tools necessary to browse the Internet. This included the HyperText Transfer Protocol (HTTP), which is the protocol used to request and receive web pages. The HyperText Markup Language (HTML), which is a markup language that describes how information is to be structured on a given web page. The first web server that could deliver web pages, and he built a combined web browser and editor that was able to interpret and display HTML pages. By 1993, CERN declared that the World Wide Web would be open for use by anyone [1]. This same year, the first widely known graphical browser was released under the name Mosaic [2], which would later become the popular Netscape browser. Later in 1995, Microsoft would release their compelling browser Internet Explorer, leading to the first "browser wars" where each one would try and add more features to the Web. [1] Unfortunately, new features were often prioritized in favor for bug fixes, leading to unstable and unreliable browser behavior. Example outcomes were the Microsoft's Cascading Style Sheet (CSS)[3], which is a language that describes how the HTML elements should appear in the browser. And also, Netscape's JavaScript [4] was developed to add dynamic behavior that could run in the browser. Microsoft created a replicated version of JavaScript, which they named JScript.[5]

2.2.2 The Early Days

In the mid 90's, web sites were mostly **static**, meaning that the documents received from a web server were exactly the same each time it was requested. This was only natural, as the majority of web sites were pre-generated HTML pages with lots of static content, e.g a company's, or a person's home page. Later, however the need for user input became apparent as applications like e-commerce would require two-way communication. User input was not part of the first version of HTML (1.0), which led to the development of HTML 2.0. This standard included **web forms**, which allowed users to enter data and make choices that was sent to the web server. The development of web sites grew into becoming **dynamic** web pages. This means that the server responds with different content depending on the input received in HTTP requests. To enable this, there has to be a program running in the web browser, that can evaluate the HTTP request, and generate a proper HTML page depending on the request itself, and the application's state. This is called *server-side dynamic page generation* [61, p.691]. Another common scenario is *client-side dynamic web page generation*, in which a program is sent to the browser, and executed inside the browser. Examples are JavaScript programs that can achieve dynamic behavior without consulting the server, and applets, which are programs that are compiled to machine code on the client's

[1] The technology behind the first version of Microsoft's browser was also based on Mosaic. Microsoft was became licensed to use Mosaic in 1995,[45] which lead to Internet Explorer 1.0.

machine and executed inside the browser. Since applets are compiled to machine code, they execute faster then JavaScript, and therefore, such technologies has for long been favored for implementing performance demanding behavior in the browser. Examples are Java applets, Microsoft's ActiveX,[6] and Adobe's Flash.[**flash**] Java applets are executed on the Java Virtual Machine, which is a cross-platform solution and hence portable to "all" execution environments. Therefore Java applets have been a highly popular way of achieving interactive and dynamic behavior in browsers. Examples include browser games, movies and 3D modeling applications.

The Problem with Client-side Technologies

Java applets and Flash had become popular choices for client-side dynamic page generation by the year 2000 [56, p.2-3], and they still exist in many web applications. There are many problems with this approach however. For instance, a plugin is usually required for running such applications inside the browser, and developers need to know an additional programming model, the user interface tend to look different then the rest of the HTML page, and on top of this there has been numerous examples of security violations with the technology itself [61, p.875-877]. Choosing JavaScript primarily would be a preferable solution to client-side interactivity, because it doesn't require an additional programming language or runtime environment considering JavaScript is already supported in all popular browsers. Having one client language is preferable to facilitate a common development model for all web applications. Unfortunately, this technology has also had its issues ever since it was introduced. Partly because of its buggy implementations due to the scurrying development processes in the early browser wars, which has lead to different JavaScript interpreter implementations by the various browser vendors. But also because browsers have not had the ability to execute JavaScript fast enough to enable satisfying dynamic behavior. For this reason, JavaScript has for long been used as an add-on language for HTML to perform simple roll-over effects, input validation and pop-up windows.

However, a lot of work has been done to provide a standardization of the JavaScript programming language. And lately, browser vendors like Google[7] and Mozilla[8] has improved the engines that executes JavaScript to enable the execution of performance demanding processing jobs.

2.2.3 Modern Web Applications

With the increase of performance capacity in modern web browsers, a lot of work has been done to standardize the development of the Web applications, such that cross-platform applications can easer be built and be supported in all browsers. Examples include the work on the newest version of HTML (HTML5). This dramatically simplifies the development of dynamic and interactive client-side behavior and media incorporation without the need for additional plugins. Also, the work on improving

the JavaScript language itself has made it the assembly language of the web, and is now one of the most popular programming languages in the world [9]. With this trend towards client-side development, the web has seen an expanding growth in web sites with rich user interfaces and lots of interactive behavior. The technologies has enabled developers to build web sites that looks more like desktop applications with a responsive user interface. Such applications are often referred to as **"Web apps"**, where the browser is the main execution platform. We will continue using the term *Web app* in this thesis, to refer to interactive and responsive Web applications with rich user interfaces.

The Social Web

In addition to interactivity and responsive behavior, there is another trend that is a increasingly becoming a key factor in modern Web apps; namely social interactions. Many modern Web apps base the information content that makes up the site on what the users adds to the page. Usually this includes users posting blog posts, comments, images and other sorts of data information. And in addition, the users connect to each other in a "social network". Popular "pure" social network applications are Facebook,[10] Twitter,[11] Pinterest [12] and many others. In addition to Web apps that serves primarily to connect users, many other types of applications like e-commerce and e-learning adds social networking ability to their web sites as a way to attract more users, and to get better user-data to improve business value. Examples include Airbnb,[13] Amazon,[14] (both e-commerce) and Coursera (e-learning). [15]

This behavioral trend has become highly popular, especially considering the users accesses the applications not only on their desktop or laptop machines, but also on smart phones, tablets, TVs, game console etc. Many people accesses their favorite social networking sites up to multiple times a day. This results in huge numbers of simultaneous users accessing their social Web apps, leading to high network traffic on the servers that hosts these apps. The scalability requirements are massive, enforcing software engineers to design for highly scalable, and efficient backend solutions.

Applications that incorporates social networking features and lets users add content naturally leads to large quantities of persisted data. This has led to the term "big data" [16] which describes such extremely large data quantity scenarios that requires highly efficient and extensible persistency systems. Many new database management systems has lately been introduced to the web marked, since the need to persist large data quantities has increased massively the last few years. Such technologies often have in common that they don't follow the traditional relational database structure (I.e SQL-based), but adopts other less structural approaches. Such databases are commonly being referred to as NoSQL.[17] The reason they don't adhere to the traditional relational structure like SQL is that this technology has showed not be fairly suited to be distributed over many database servers, e.g in cloud environments. [47] This is a severe problem, because the ability to scale over many machines is not only feasible for load balan-

cing, but an absolute requirement when extremely large amounts of data must be persisted. Most noSQL databases on the other hand, has showed its ability to scale very well over multiple servers, making it a good choice for persistence solutions, especially when web apps are deployed in the cloud.

A final note with the Web applications just described is that they often offer their services to external third party clients through, what's commonly called their "public API". This means that other external applications, (e.g some other Web app) might use functionality that the app is offering publicly as a service, and incorporate this functionality into their own app. This is called a service-oriented architecture.[54] An example is that some Web app A uses geographic data that they fetch from the public API offered by Google Maps.[18]

The applications described in this section are commonly referred to as Web 2.0. [60] The term has a widespread meaning and a lack of consensus on its true definition. However, in this thesis we will use the term Web 2.0 as Web apps that incorporates social networking and public API offerings.

2.3 Web Technologies

Having looked at how the Web started, and given an overview of common attributes and features for Web 2.0 apps, we will now focus on the technologies that hosts these applications. In this section we will discuss the main technologies that are used to build and host modern Web 2.0 apps. First, an introduction will be given on the client-side technologies that executes the Web apps, then we discuss some common architectures and principles for designing these Web apps.

2.3.1 Web and Application Servers

Also called HTTP server, is a program running on a dedicated server machine, that hosts web content to client users. The client is usually a web browser, but it could also be a web crawler, who often intends to gather information on web pages searching purposes. The web server manages HTTP communication with the client users, and serves static content like images, videos, or stylesheet files. Examples of commonly used web servers are Apache Web Server,[19] and Microsoft Internet Information Server.[20] The web server is responsible for delegating requests for dynamic content to an **application server**. The application server hosts the web application itself, also called the backend, [2], and hides the low level details of HTTP requests, typically by wrapping HTTP-header info into separate programming language variables. The application server can route specific URL requests to appropriate handlers in the web application. Application servers usually supports one or more **web application frameworks**, which simplifies the development of a web

[2]The application's **backend** is the application-code that runs on the server, in conjunction to the application's **front-end** which concerns the code that runs in the browser.

application in a specific programming language. Examples are SpringMVC for Java [57], Ruby on Rails for Ruby,[21] and Django for Python.[22]

2.3.2 The Web Browser

Browsers are software applications that requests and displays content on the Internet. The information is usually expressed in HTML pages, but it can also be other types of data for instance images, script files, PDF files, or videos. The way browsers interprets the data is specified by World Wide Web Consortium (W3C),[23] however up until recently, the various browser vendors have usually not completely conformed to the whole specification but instead developed customized solutions. This has caused many compatibility issues for web developers.

High-level structure

The browser's software stack consists of a set of components that each has individual responsibilities, and cooperates with the work of fetching and displaying web resources. The main components are listed below :

1. User interface
2. Browser engine
3. Rendering engine
4. Networking
5. JavaScript interpreter
6. UI backend
7. Data persistence

The rendering engine is a very important part in the process of displaying a resource. Its responsibility is to get the document from the network layer (usually 8 bits at a time), render the document and finally paint the result on the display. The process of rendering the document is showed in figure 2.1 on the facing page. Note that this process is iterative and will happen repetitively until the whole HTML page with all its external resources are completely processed. The rendering engine's lifetime is **single-threaded** and runs in an infinite loop that listens to events. An event might be to calculate a new position of an element, perform painting on new or modified HTML elements or handle a mouse click. However, if multiple external resources are to be fetched at the same time, the browser can, and often will create multiple HTTP connections that will run in parallel to efficiently load content that needs to be contained in the main HTML document.

WebKit[24] and Gecko[25] are two popular rendering engines that implements the rendering process in the figure, however they do differentiate slightly in their internal behavior. WebKit is the engine that runs Chrome

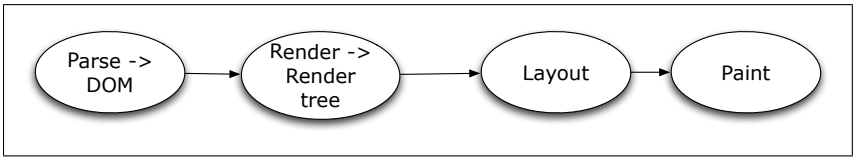

Figure 2.1: The rendering engine's responsibility

and Safari, while Gecko runs Firefox. In this section we limit the discussion to concern only these platforms, as they are built upon open source solutions and hence have available technical descriptions. The text that follows describes the process of rendering a complete HTML page. This usually happens when the client-browser requests a URL for an HTML page and the browser "refreshes" the page with the new content.

Parsing The rendering of an HTML page starts when the networking layer is instructed to fetch a URL, say *www.google.com*. As the HTML page is fetched, the browser will immediately start fetching all external **links** that are contained inside the HTML page. This could be links to stylesheet pages, JavaScript pages, images, videos, etc. As soon as the initial HTML page is received in the networking layer, the rendering engine will start requesting chunks of data. Then it will start and **parse** the HTML and stylesheet documents. An important feature of the HTML document's structure is that all the markup elements (tags) are nested in a hierarchical structure. Hence, when the HTML document is parsed, its tags are laid out in a tree structure, and if an error is found in the page's structure or syntax, an error is thrown. Note however that the HTML parser is "forgiving" in that it will fix errors found in the HTML document.

Whenever the parser hits a script tag, it will first fetch the script if it is referencing an external file. Then, it will execute the script immediately. Unless the tag is marked "deferred" in with case the handling of the script will be postponed until the HTML parsing is done, the parser has to wait until the script is both fetched and executed. This is because the script might try to manipulate the DOM tree. To improve performance by avoiding the HTML parser to block while scripts are loaded, scripts can also be marked as "async" in which case the modern browser will to generate a separate thread that fetches (if needed) and parses the script asynchronously in a separate thread, while the main parser thread can continue parse HTML.

The tree that is generated is called the **DOM tree**, where each tree element is named a DOM element. When the whole HTML page is completely parsed, the rendering engine will start executing the scripts that are marked "deferred". When these scripts are finished executing, the browser will generate a **DOMContentLoaded** event. Finally, if there are any depending scripts that were marked "async" that are still being fetched or executed, and when all external resources are fetched, the window's **load event** is generated.

Rendering While the DOM tree is being populated, the rendering engine will also start generating the **Render tree**. This is another tree that is a visual representation of the DOM tree, and in effect decides the style and order of how the DOM elements should be laid out. Every element in the Render tree has a reference to its DOM node, a style, and in addition they know how to layout and paint itself and its children. In effect each render element represents a visual **rectangle** on the screen. However, it is not necessarily a one-to-one mapping between a DOM element and its render element, because some DOM elements may need multiple rectangles on the screen, and hence have multiple render elements attached to it. The rendering tree is generated from all the styling elements contained in stylesheet documents, styling information in the HTML document, and default styles provided by the browser. While attaching render elements to DOM elements; if the stylesheet is not fully loaded yet, it will save a "holder", so it can continue later.

Layout In the layout process, each render element is given coordinate instructions for where on the screen it will be placed, and the sizes it will be given. The calculation is performed recursively from the root HTML node to the bottom. Also, the layout process can be *global* in which the entire rendering tree performs layout calculation, or *incremental* in which case it uses a dirty bit system; each time a new render element is added or modified it is marked dirty. When an incremental layout process happens it will calculate only the render elements that are marked dirty.

Painting The painting process does the actual work of painting the elements on the screen. It does so by iterating through the rendering tree and paints each component. Like the layout calculation, painting can also be done globally on the whole rendering tree, or incrementally. In the latter case, whenever a render element is changed in a way that does not affect the entire tree, the renderer will cause the OS to recognize the changed renderer and hence trigger a **paint event**.

2.3.3 JavaScript

JavaScript is an interpreted programming language primarily built for manipulating web pages. The language was developed at Netscape in 1995, during the time when Netscape and Microsoft were battling for the majority of browser users. The language itself was built in only 10 days, by Brendan Eich.[50] He had instructions to develop a language that would look like Sun's Java, only simpler, interpreted and easy to integrate into web pages, so that it would appeal to non-professional developers. The reason was that at the time, Netscape was built with Java, and the browser-server model was considered a distributed application environment supporting cross-platform development. The only problem was that HTML wasn't sufficient to support complex programming operations, and Java was a rich, and comprehensive language aimed

for professional developers [50]. Hence the need for a hybrid solution. Eich designed the language to follow much of the syntax from the C programming language, only simpler and a much more dynamic memory management system. At the time, a web page's lifetime would usually last from a few seconds to a couple of minutes, so therefore the language had only a simplified concurrency model and memory management [50].

Despite a considerable amount of buggy features in the language and some compatibility issues between the different browsers, JavaScript quickly became a highly popular language for the web. Developers easily create interactive behavior like changing color on a button when a mouse hovers over it, give the user feedback if the input in a textfield is wrong etc. Much of its success was because of its simplicity; there was a low barrier to add JavaScript behavior into web sites. Because there is no compilation process, and no "start function", only independent functions that can easily be written and "tossed" around in the document, unprofessional developers could quickly implement exciting and dynamic features [26]. On the other hand, for this reason, many professional developers would consider the language of being strictly for amateurs and not suitable for professional developers [27]. Its strength, however, was clearly for small sized applications, as browsers at the time were not able to execute large-scale JavaScript code. Also, considering that the JavaScript language at the time was very simple and limited, the development of large-scale JavaScript Web apps was simply not feasible.

In an effort to improve the language features of JavaScript, and its browser incompatibilities, a standardization process of JavaScript was given to the European Computer Manufacturers Association (ECMA) in 1996. The language was actually renamed to ECMAScript, although most people still refer to it as JavaScript [**jsHist**]. Unfortunately, as the standardization was being developed, the browser inconsistencies, especially between Netscape and Internet Explorer continued to grow. Many JavaScript frameworks were built as workarounds to the inconsistencies, but most solutions weren't good enough. This led to alternative solutions for highly interactive and graphical client behavior such as Adobe Flash or Microsoft's Silverlight. [28]

An important part of JavaScript's history was with the rise of **Ajax** technologies,[29] which gained much attention right around a century after JavaScript was first introduced. Ajax is an API that offers JavaScript functions that makes the browser asynchronously fetch data from the server without having to refresh the current web page. Instead, the requested data would be used to manipulate the web page. This would result in a much more interactive user experience because the browser would neither block while waiting for the request, or re-render and paint the whole DOM tree upon successful complete. Since the introduction to Ajax, JavaScript has increasingly become a highly popular language, and it has brought the attention of professional developers. This has led to successful framework solutions like JQuery[30] and Prototype,[31] which simplifies the development of complex dynamic behavior and fixes browser incompatibilities, so that the web developer doesn't have to write

specialized code for each browser.

The increasing popularity of client-side application development with JavaScript has led to powerful JavaScript engines in modern browsers, making memory management and JavaScript interpretation highly effective. In September 2008, Google built the Chrome browser with its V8 JavaScript engine,[32] stating that low performance JavaScript implementations are no longer sufficient. Other browser vendors followed along, and today JavaScript performance is more superior than ever. Still, however, there are drawbacks with the language itself. Even though libraries like JQuery and Prototype simplifies the development of interactive and cross-platform web pages, it is easy to end up with a big pile of tangled JavaScript event handlers and unstructured functions. Most of the reason is that the language lacks features like classes, modules and namespaces, which makes it difficult to develop flexible and maintainable large-scale JavaScript applications. However, a lot of work has been done lately to implement quality frameworks that provides comprehensible syntactic sugaring for the language. These frameworks use the nice concepts of the JavaScript language like inheritance and closures to enable a highly flexible and structured development environment. All this has led to a massive amount of large-scale JavaScript web applications where much of the backend code has been moved to the client tier, implemented in pure JavaScript. Good examples are Google's Gmail,[33] and Maps. [34]

2.3.4 Client-server Interaction Schemes

There are multiple ways the browser can communicate with the web server. In this thesis, we mainly adhere to two different ways ways: synchronous client-initiated requests, and asynchronous client/browser-initiated requests. These communication schemes all use HTTP, and are client-pull based. However there are other push-based alternatives like Comet,[53] or WebSocket,[35] where the client and server maintains an open connection, and the server can notifiy the client of changes.

Synchronous

In a synchronous HTTP request, the client-browser asks the server for data, in which the browser will wait for the server to respond. The request is normally trigged by the user who writes a URL in the browser and presses enter, or an HTML form that is submitted. This could be a button that is pressed inside an HTML form tag, the enter key is pressed inside a text field that is contained in a form tag, or a link that is clicked on a given page. The server usually responds with a new HTML page that is sent to the browser, in which case the browser would start a complete rendering process to build a new DOM tree, and paint it on the screen. A normal term for this is a page *refresh*, or *reload*.

Asynchronous

In asynchronous HTTP requests, the client-browser sends a request to the server and continues to execute without waiting for the server to respond. The request can be initiated by the user, or the browser might trigger the request itself. When the server has completed handling the request, it will send the respond to the browser, which is then notified and will handle the result. Usually in this communication scheme, the result is not a complete HTML page, but rather a more fine-grained data object. In this case the browser would perhaps manipulate the DOM tree to alter the page content in correspondence with the result that came from the server. The browser does not have to reload the page, since the browser's data structures (DOM/rendering trees) does not have to be re-built. The client user is usually unaware of the request, because the browser doesn't block while the communication happens.

The most popular implementation of asynchronous communication with HTTP is Ajax. An Ajax request can be sent to the server, in which case the browser does not have to block while the request is being processed. When the request finishes, and the browser receives the response, the browser will initiate an event, in which the developer would have created an event-handler that parses the result, and performs some action based on the result.

One potential drawback with Ajax is that not all browsers supports JavaScript. Examples are some smartphone devices or PDA devices. Also, some old browser implementations would not let Ajax requests send the browser to a new *state*. This means that if the user hits the back button after an Ajax request has been performed, the browser would go back to the last full-page reload that was accessed, instead of going back to the page as it was right before the Ajax request. Last, pages that are generated using Ajax are not automatically picked up by web crawlers, because most web crawlers does not access JavaScript code. This means that content generated by Ajax would normally don't show up in public web searches.

2.3.5 Input validation

Protecting the application from malformed user input is an important aspect of web applications. Not allowing clients to submit illegal input both helps making the users sure they entered the data they intended (humans often makes mistakes), and it protects the application against malformed data that might have bad intentions. Even though the HTML pages might have validation logic built into the page (e.g by JavaScript), there are ways to manipulate the data that is sent with HTTP requests. Examples include SQL-injection attacks [62], or other types of HTML form injections [**htmlinjection**], in which the attacker is able to alter the execution of a given form. Therefore, user input should be performed both on the client tier for quick validation feedback, but also on the server, in case the client tries to override the validation in the Html page.

2.3.6 HTTP Sessions

An HTTP-session is a semi-permanent communication dialogue that exists for two communicating entities (here, the client and the server). A session normally has a time-out value, such that when the time runs out, the session ends. The HTTP protocol is stateless in its nature, because every HTTP request is self-contained, and independent of every other request. Therefore, to be able to maintain state in an application, the client and server has to incorporate a session protocol. The state information itself is data that has to be maintained between multiple pages in the application. Examples are shopping cart information in an e-commerce site, flight booking details, or authentication credentials. Imagine the user having to identify himself for each request he sends to the server. This can be avoided if state information is persisted and being referenced for each request.

There are a couple of ways to implement sessions. State can be kept and managed on the client, on the server, or both by having state information inside the messages sent between the client and server. As for server generated sessions, an implementation is usually provided by the web application framework that hosts the particular application. In this case, the server generates a session-object that will contain session data. This object can be kept in-memory, in the database or stored in flat files. To identify the session object when a request comes into the server, some web frameworks choose to put a unique id in the HTTP cookie header-field, or if cookies are not supported by the browser, the identifier can be injected inside URLs. The session identifier is then included in every HTTP request the user sends to the server. An example is given in figure **??** on page ??. Note that the session object can, and often is used as a cache for data that the given user would access often. That is if the session object lives in the server's memory and not in the database. This avoids having to fetch the same data from the database needless times.

State information can also be maintained inside the messages themselves, that are sent between the client and server. This could be to place the data inside hidden fields in HTML forms, store the data in cookies, or store the data in the URL. However, this approach is not very flexible, as the data has to be manually coded into all the places in the HTML pages where requests are sent to the server, plus that the data size is limited, as is the data format.

A third alternative is to store session state in the client's browser. This works in the same way as the server-side approach, in that a session object is stored in browser memory or some external database managed by the browser. The advantage of this approach is that no session identifier has to be included in the HTTP requests sent to the server, since everything is managed by the client. If the session is to be stored in the browser's memory, the browser has to support HTML5, which provides a programming interface (API) for communicating with an in-memory database in the browser.

2.3.7 Representational State Transfer

In web applications, each individual HTTP request is handled by a particular function that resides in the application's backend. Modern web applications often follows a design pattern named Representational State Transfer, or simply REST. [49] This pattern states that all HTTP URL's must reference a particular resource on the backend, by using one of the four supported HTTP operations Get, Post, Put, or Delete. This way, every URL offered by the web application are self-contained in that it contains all the necessary information needed to satisfy a request. Resources are identified by a URI, which uniquely identifies a resource, and one of the four HTTP methods that should be performed on the resource. Applications that follows this pattern are named RESTful applications. A common use case for RESTful applications is to offer the self-contained URL's publicly as an API to clients other then just the application's own front-end, like other third party applications that wishes to use the applications REST services. Further, the REST pattern states that each REST request is stateless, hence adhering to the nature of HTTP which is stateless. That means, REST requests should not depend on an ongoing session in order to generate proper results.

2.3.8 JSON

JSON[36] is a text based data format that is based on a subset of the JavaScript programming language. It is easy for both humans to read, and machines to parse, and has a similar syntax to many of the programming languages based on the C family of languages. The data structure fits well as a transmission format in web applications, because it is both simple and light weight, easy to modify and is supported by many programming languages. The format itself is very simple, and the datatypes offered are limited to numbers, strings, arrays, objects (key-value pairs) and booleans. An example of a JSON object representing a guitarist is showed below:

```
1  {
2    "username": "Paul Swayer",
3    "age":21,
4    "country": "Norway"
5    "guitars":
6    [
7      "Gibson Les Paul",
8      "Fender Stratocaster"
9    ]
10 }
```

This object contains two string key-value pairs, one integer value, and one array consisting of two simple string values. Notice how the object itself is made up of key-value pairs. RESTful applications might choose to use JSON as a transmission format. For instance a JSON object can be sent with a PUT request, so that the REST receiver will update the object with the content in the JSON object. Also, in a REST-get request, the receiver can return a JSON object to the client.

The closest alternative to JSON is XML,[37] which is another transmission format often used on the web, and especially with REST communication. However, its syntax is a bit more verbose, and requires more processing to manage because of its markup tags and syntax rules. On the other hand XML lets one add more restrictions to the data then with JSON.

2.3.9 Template Rendering

In dynamic web pages, when an HTTP request comes in for a particular HTML page, the server has to prepare and populate the HTML page with proper data variables before the completed page is sent back. Normally the server inspects the HTTP request that comes in by looking at the HTTP request headers, the content of the HTTP body, and a session identifier if it's provided. Given the information provided in the request, the server will perform an action, and probably fetch data from the database. This data is to be injected into the HTML page that will be returned to the client.

One way to generate an HTML page that contains the newly generated content, is to generate the HTML directly in code as Strings, and send the result back to the client. However this approach is messy, difficult to maintain, and the developer has to know the programming language that is creating the HTML strings. In other words, not a preferable solution for web designers who only knows HTML. The preferable approach is to use a template system. A template system consists template files (often called views), which are files that consists of HTML with special markup code that refers to and can operate on the data variables in the web application. The operations supported are usually limited to simple loops and conditional expressions, just enough to facilitate the injection of dynamic data without confusing front-end designers. The template system also has a template rendering engine that takes a template file, the data to populate the file, and produces an HTML page. This way, when a client request comes in, the server will generate some data, probably from the database or an in-memory cache, create an object that is filled this data, and send the object together with the proper template page into the template rendering engine. The resulting HTML page is sent back to the client.

Template rendering can also happen on the client. The process is pretty similar. Rendering is performed by JavaScript code that "lives" inside the HTML page. This is done by adding special syntax in the HTML page that is recognized by a rendering engine, and ignored by the browser. When some HTML page is to be rendered, the JavaScript rendering engine is called with an HTML page and a data object as input, and it would return the same HTML page, but altered with the new content from the data object. Unlike server-side page rendering, there are no special file type for template files. They are just HTML pages with special syntax that is ignored by the browser when the HTML page is rendered. Hence server-side template rendering generates a new file given a template file, while client-side template rendering only modifies an existing HTML file.

2.3.10 Databases

Storing persistent information is an essential part of web applications. The content that is offered must be able to be saved, retrieved, deleted and modified in an efficient and flexible manner. Databases, and especially relational databases has since the beginning of web application history been the most popular form of storing persistent data.[**engineering**] Other alternatives have been used like flat-file storage (where the content is stored as plain text or binary data) or XML- or object databases, but ever since relational database management systems arose back in the early 1970s, it has been a highly popular form of data persistence. Much of the reason is that it provides **durability**, which in the context of data persistency means that once the data is stored, it is guaranteed to exist even if machines holding the data crashes. Also, a reason for the relational databases's popularity is that it stores information in a structured format, which often fits the structured data formats that are manipulated by the web applications. However, as the extent of information being persisted in modern web applications has become incredibly large, other types of database technologies has also entered the marked lately. These are commonly referred to as noSQL databases.

ACID

ACID is a popular term in the context of databases. It is a set of properties that guarantees reliability when it comes to transaction management. Database management systems often state that ACID guarantees are provided in their system, in order to promise a reliable database solution. Each property is defined below:

Atomicity guarantees that either all the commands in a transaction completes, or non no.

Consistency guarantees that all the data will always be in a consistent state according to pre-defined rules. A transaction brings the system to a new consistent state.

Isolation guarantees that parallel transaction executions are always processed as if they happen serially, i.e no interference of any two parallel transactions.

Durability guarantees that committed transactions are safe, and lasts even during system errors or crashes.

ACID guarantees are often provided by relational database management systems (RDBMS). However, noSQL databases tend to have a more relaxed relations to the principles, in favor for speed and scaling abilities.

CRUD operations

In web application terminology, one often use the word CRUD to refer to the four essential database operations; *Create*, *Read*, *Update* and *Delete*. These are operations performed by the web application, on the data that needs to be persisted. When a CRUD operation is executed, it is the application's responsibility to convert the data into a format that fits the database's technology, and vice-versa. This process is called marshaling, or serializing. One example is when the application is to save a new object in the database. In this case, a *Create* operation will be performed where the application will transform the object from whatever programming language syntax the object is currently described in, into a structure that fits the given databases' syntax. The application will send this transformed (marshaled) object to the database, which is now able to parse the object and save it to its storage structure.

Relational databases

Relational database management systems is a storage system based on a formalism known as the relational model. The formalism is based on structure and relationships, where the data entities are stored into **tables** that contains a set of **attributes** that describes the table. The tables can be related to each other to form groupings. RDBMS's stores a collection of tables, where each data entity is represented as a **row** in a specific table, and each column in a row represents an attribute for that entity. The most popular form of manipulating data in a RDBMS is SQL (Structured Query Language).[52] This is a query language used to insert and manipulate data in a relational database. There are popular dialects of the language, generated by database vendors like Oracle's SQL,[38] Microsoft's MS SQL[39] and the open source product PostgreSQL.[40]

NoSQL

NoSQL is a broad class of various database management systems who all have in common that they don't share the relational structure from normal SQL databases. The reason for its existence starts with the rise of Web 2.0 applications, when developers saw the need for simplifying replication of data, higher availability, and a new way to manipulate data that can avoid the need to perform tedious mappings between SQL strings and objects in any given programming language. The main potential for noSQL databases is to perform operations on massive amounts of data that is not structured or connected in complex relationships. Very often this applies to Web 2.0 applications, because much of the information in such applications can be persisted as nested key-value data, such that the values can be arrays of key-value data. A typical example is users that has arrays of blog posts, and blog posts has arrays of comments, in which case the users and blog posts could be identified by a name (key). There are many different classifications of noSQL databases, which vary in the way they structure

the data. An overview of the some commonly used noSQL categories is summaries in the following list:

Key/value store
: Is a simple database store where data is identified by a key, and the data itself can be any datatypes usually supported by the implementing programming language. The structure is schema-less, meaning it doesn't provide complex structures with foreign key constraints. It is also highly efficient as the database is often implemented as a HashMap. One popular example is Redis. [41] This is an extremely fast key-value store that favors speed over durability (the guarantee that a committed transaction lasts forever). It also provides simple replication support, making it easy to distribute the database over multiple machines. Much of the reason for Redis' extremely high speed is because the data is typically being kept in memory, and only written to the database as a snapshot every once in a while.

Document-oriented databases
: Is a datastore that is based on documents that contains unstructured content. Documents are often separated into unstructured collections (can be viewed upon as SQL-tables), where unstructured here means that content in the same collections can have different structure. However there is some variation in the way the different database implementations choose to define the formats of the documents, but it can be assumed that each document encapsulates some logically associated data in a predefined format. An interesting property with these databases is that performance is often not the main goal, but rather programming satisfaction. As many of these are implemented in JavaScript and offers querying semantics and data structures based on JavaScript objects, it is really easy and flexible to perform database operations on them. Examples include CouchDB[42] and MongoDB. [43]

Column-oriented databases
: Is a database system where data is organized as columns, as opposed to row-oriented databases such as SQL based databases. In this scenario, every value that would usually be in a row gets its own instance in a column together with its belonging identifier (Id). As such, it is very efficient to perform range queries over a big amount of column data. Examples are Cassandra[44] and Google Big Table [51] (although these are not pure column-oriented, but rather a hybrid). ...

2.4 Summary

We started this chapter by looking at how the World-Wide-Web began in the late 80's. In the beginning, web sites were primarily static web pages,

but gradually turned into more dynamic pages where the content changes depending on attributes provided by the client. After this we saw that some of the technologies that are found on the Web today are influenced by the browser wars that started in the mid-90's. This especially concerns the JavaScript programming language, which has resulted in many browser incompatibilities and a somewhat misunderstanding and unawareness of JavaScript's core language features and capabilities. However, major browser vendors, and Google especially, has acknowledged the advantage of having a unified language for the Web, something that has brought a lot of attention and improvements to the JavaScript programming language. This has now made JavaScript become a highly popular language for the Web, and developers have started building large-scale dynamic Web applications purely in JavaScript.

In the second part of this chapter, we looked into popular technologies that are commonly found in modern Web applications. Examples included Ajax, which offers an asynchronous client-server communication scheme, Rest, which is a design pattern that states that Web resources should be manipulated through exclusively through methods specified in the HTTP protocol (GET,PUT, POST and DELETE), and noSQL databases, which are databases that doesn't abide to the relational model, but instead specializes on speed and scalability through replication. We also looked at a specific type of modern Web applications, namely Web 2.0, which we defined as interactive Web apps with responsive and rich user interfaces, social networking features and public API offerings.

Chapter 3

Design Alternatives for Modern Web Applications

3.1 Introduction

So far we have discussed the behavioral trends in modern web applications, and some common technologies that enables such applications. We saw that modern web applications are often very interactive with rich user interfaces, and that these web sites looks and performs like native desktop applications with graphical user interfaces. In addition there is often a requirement for a high number of simultaneous users, and a lot of data is being stored and manipulated. In this chapter, we look at the various technologies that are commonly used to design and build such interactive and scalable software applications. This concerns the application's **architecture**. The application's architecture describes the concepts of the application's structure, at different granularities. In this thesis, we will discuss architectures both seen at a high-level perspective, and also down to low-level source code perspectives.

As the behavioral requirements for web applications has increasingly changed the last years, so has the application's software architectures. The traditional web application has in the last decade followed a three-layered architecture where all the logic happens on the server. This is called a fat-server architecture, where HTML pages are rendered on the server and handed to the browser every time the client issues an Http request. Lately however, there has been an increasing interest in moving much of the applications logic to the client, and abandoning the server-side page rendering in favor for client-side page rendering for dynamic HTML content. This is called a thick client architecture. [1] Still however, a lot of applications follows the traditional approach, as many developers and

[1] The thick client architecture is not a new idea; thick client architectures has been around for many years, where programs are sent from the server and executed in the client. Often this would be online games, calculators, or other user-interactive applications. However, these applications depend on a specific program that is compiled and executed on the client, in other words, not traditional web pages that uses common browser-supported technologies.

application owners are skeptical to the fat-client model. This architecture implies heavy use of JavaScript, which has since its beginning had a lot of opposition, and there are still many browsers that does not fully support JavaScript behavior.

The main purpose of this chapter is to outline the differences between the traditional web-app architecture, and modern web-app architecture. We will look at common trends and popular solutions in how developers often design and implement the respective architectural models. The first part of this chapter is divided into two main sections, one for each architecture. After this we will explain how these solutions have been deployed in the prototypes built for this thesis. For simplicity purposes, we will refer to the traditional approach as **Reference-model 1.0**, while the latter approach will be referred to as **Reference-model 2.0**. Hence, these two will be our reference models for software architectures that are aimed to build modern, interactive and scalable web apps.

3.2 Traditional Web Application Architecture

Going back approximately 15 years, web applications where often built with CGI technology using tools like Perl and ColdFusion [**webstart**] . With the CGI technology, a web server accepts URLs that are delegated to an appropriate CGI program. A process is started on the server, and the CGI program executes the given request, which results in an HTML page that is sent back to the client. This solution however, was not very scalable considering each request would trigger a new process on the server. Gradually, as the web got more users and the applications became more complex, new web framework technologies came along. Examples are PHP, J2EE, Ruby on Rails and Microsoft's .NET [**topframeworks**]. For many years, developers have been building web applications with these technologies, where all of the application's business operations are executed on the server. This implies that the backend implementation has many responsibilities, and the front-end is simply a thin client that doesn't need to do much processing. This is only logical, as backend implementations runs on powerful web servers, and client devices has up until recent years not been able to perform demanding processing jobs.

In an HTTP request, the backend application receives requests (often named an **action**) from a client which is handled by a **request handler**. The handler performs validation of the input data, executes the necessary business logic, and if necessary, manipulates the database. To be able to serve multiple simultaneous users, the application server often generates a separate thread for each incoming request. At the end of the handler's execution sequence, the application prepares an HTML page that it sends back to the client. The client tier refers to the front-end code that runs in the client's browser. It consists of an HTML and zero or more CSS files. The modern web application proposed in this thesis requires a lot of interactive behavior which is usually best solved with JavaScript. Other popular alternatives to JavaScript are for example Flash, Microsoft Silverlight, or

ActiveX. However, these all have in common that most browsers don't support them out of the box, meaning a plugin has to be downloaded and installed in order to use any one of them. Also, these technologies tend to provide their own graphical user interface, so that when they are integrated into the web app, they tend to look different then the web app's own "style".

The HTML file contains just the necessary JavaScript code to enhance the user experience. The script code is either embedded in the HTML file inside *script* tags, or it is referenced as external JavaScript files that must be fetched by the browser. The JavaScript code is often a collection of **event-handlers**, which are independent functions registered to be notified when a given event happens in the browser. Examples of events is when a mouse is clicked that will start an animation, the mouse is hovering over an image, a button is clicked that will open a small pop-up window, or some text is entered that needs quick validation. The event-handler will look at the current state of the web page, execute some necessary JavaScript routine, possibly making an Ajax request to the server to get some data, then manipulate the browser's DOM tree to update the page. Note that the JavaScript event handlers are only used for small-sized user events that needs quick results. User actions that leads to bigger changes, like URL- or form requests are not handled by JavaScript but are normal HTTP requests sent directly to the server, leading to a new HTML page sent back to the client.

3.2.1 The Three-Layered Architecture

A classical way of separating concerns in a web application is to divide the whole system into three different software layers. This is called a three-layered architecture. There are some variations to how these layers are separated, but in this thesis, when we refer to the three-layered architecture, we follow the layering structure outlined by Brown et. al.[**brown**] This architecture separates the system into a Presentation layer, domain layer, and a data source layer. These layers reside exclusively on the backend of the application. The front-end (e.g the code that runs on the client) has little responsibility in this architecture, as its only task is to display the result that is produced on the backend. The layers are designed to be very loose coupled. This is done by avoiding that a module in one layer depends on a concrete implementation in a lower layer. Instead, they depend on abstractions (interfaces) which can easily be swapped out **??**. The abstractions are not hard-coded into the layers, but are "injected" through function arguments so that the same function can be called again if one wants to change the type of a dependency. The benefits from having individual implementation details encapsulated in different layers, is that it is very easy to modify one layer without harming another. And in addition, layers can be reused by other software modules. For instance, the data source layer could be re-used by another module that also wants to use the database, but does not want to have to go through a domain, or presentation layer.

The presentation layer is responsible for displaying information to the end-user by accepting HTTP requests, and return prepared HTML pages. The presentation layer is the main entry point on the backend, where URL requests are received and handled by a dedicated action handler. Such a handler is often called a **controller**, which is a dedicated function that is called by the application server, to handle the URL request. The controller is a central part of the **Model-View-Controller** pattern. This is an architectural design pattern that organizes the structure of the application by a logic separation of concerns. In this pattern, **Controllers** handles HTTP requests from the client, delegates to proper business operations that is handled by the **Models**, and based on the result, the controllers will create a **View** that is returned to the client. Another architectural structure is the **Application controller pattern**.[**pea**] This separates how objects are to be presented in the app, from the app's business operations, by adding a new layer which is responsible for deciding which page to show in which order. This structure is nice if there is a lot of logic required to decide the page's ordering and navigation scheme. However, as the most common architectural pattern for the presentation layer is with the MVC pattern, we will continue the discussion with this pattern in mind.

When a URL request comes in through the presentation layer, it goes through a number of filters before control is handed to the controller. Filters can have different objectives like authentication, marshaling of different web formats into an object in the programming language that is used , error handling or Html form validation. Also, the presentation layer would check the URL for a session identifier in the request's cookie, or the URL itself. If there is one, the session identifier is referencing a session object that is in already residing on the server's main memory, or it is fetched from a database. The presentation layer might validate the data parameters that are found in the HTTP request and verify that the user (identified by the session object) is allowed to perform the action it requests. After request are passed the filters, the appropriate controller function is called. The controller function usually has little responsibility, other then to delegate to the right business function in the domain layer, possibly together with some data objects found in the session object, and parameters given in the request URL.

When the business function returns back to the controller, it looks at the result to determine which view to return back to the end user. The view might be an HTML page, a page written in a template language, or another data-format like XML or JSON. The latter two formats are most often used if the request is an Ajax request, in which case the client just wants some data with no markup formatting added to it. Then the controller handles the resulting object to a marshaller which transfers the result object into either XML or JSON. If the view to return is a template file, the controller will handle the execution on to the rendering engine which compiles the template file together possibly with a data object that is being used to populate the page during rendering. The data object might be the result of the business operation (for instance some data that was fetched from the database), or some data that is stored in session object (which may have

been modified during this process). In which case the controller would send this data into the rendering engine together with the view itself. The result of the rendering process is a new HTML file that is sent back to the client user. It is also possible for the controller to send multiple template files into the rendering engine in order to produce one single HTML file as result. This comes in hand if the various pages on the web app contains multiple repeating HTML fragments. Typically this might be a navigation bar or a footer. In this case these reoccurring HTML pieces could be implemented in separate template files, and reused in the different pages. The process of rendering pages on the server with a rendering engine is often referred to as **the template view pattern**.[55, p.REF] Also, the process of creating dynamic HTML based on some application state is often referred to as **View logic**. This is a term we will use often when discussing how, and where views are generated.

The domain logic layer , also called the business logic layer, is responsible for executing the business operations that are supported in the web application. These are the functions that makes up the core services of the application. The domain logic layer should be built with an emphasis on flexible and coherent code so that the business rules can easily be extended and modified. To enable this one should design with proper design patterns and techniques like test-driven development [**beck**]. It is in the domain layer one can find the **domain objects** (or business objects) that acts as the representation of the core entities that exists in an application. Examples are a Movie, a Book order, a User, a Blog-post etc. These are often encapsulated in a coherent class, or a similar programming language pattern. *We will use the term domain objects quite often in this thesis*. The business operations in the domain logic layer uses the domain objects, by executing operations on them. Examples are calculating the total price for an order of books, registering a new friendship between two users, or searching for recommended movies for a currently logged in user.

There are multiple ways of organizing how the business processes are implemented in the domain logic layer. A simple structure is by using the Transaction Script design pattern [55]. In this structure, each business operation is encapsulated in one independent, self-contained procedure (script). Each procedure is mapped to an operation in the Presentation layer, that takes input from the presentation layer, performs business logic (e.g data validation, calculations etc) and stores data in the database. This approach is very simple, as there is no complex object-structure, just independent functions. However as applications get complex this pattern might lead to code duplication as different transactions might have common behavior. Another more common and object-oriented structure is to use the **Domain Model design pattern** [55] where business logic is encapsulated in domain objects. For example a BookOrder class would have functions for creating a book order given a list of books and a user id, fetching a book order given an order Id, updating a book order, and deleting a book order. The domain objects can have dependencies on each

other, and call each others function in order to gain code reuse.

The data source layer is responsible for communicating with other systems, such as databases, messaging systems, external web services, the filesystem etc. For instance, if the application is to store data to the database, this will happen here. Also, if the application is to talk to a third party API, for instance fetching geodata from the Google maps API[18], this will also happen in the data source layer. Reference-model 1.0 uses a relational database management system, which is the most popular solution for choosing how to persist the data in a modern web app.[**sqlpopular**]. The reason for its popularity is much due to the relational model, which brings a flexible and highly efficient query language (SQL). Most computer science courses on databases tech SQL, so developers tend to think data relationally [**sqlpopular**]. Also, data safety guarantees is provided with ACID, and the wide offerings of relational database management systems with its many development toolkits makes it a natural choice for the database solution. Not to mention, the technology itself is many decades old and hence brings years of experience and documented best-practices.

In the context of persistency, the data source layer is responsible for converting the in-memory objects into the proper storage representation, and vice-versa. For instance translating a SQL table into a Java object. The data source layer has to connect to the database, handle database transactions and close database connections. Usually this is handled by the web application framework, so the data source layer only has to worry about how to perform operations on the database. As with the domain layer, there are two popular patterns for structuring the data source layer. One is with the active record pattern [55], in which case each domain object would know how to perform CRUD operations on themselves. Another approach is the data mapper pattern [55] where there is one separate class for each domain object, that is responsible for performing the marshaling of the given domain object. For instance, if there is a class *Person* in Java, it could have a reference to a mapper class called *PersonMapper*. This class could have 4 functions, one for each CRUD operation. These function would know how to communicate with the database, execute SQL queries, and transform a Person object to a SQL table and vice-versa. The data mapper pattern separates the persistence code out of the domain objects, but adds more classes to the system. The active record pattern has a tendency to grow big in size, if the domain objects has to support a large amount of complex database operations.

There are also frameworks that hides the complexity converting between database entities, to in-memory objects. These are called object-relational mapping (ORM) tools, and are framework injected into the application that creates an in-memory version of the database. They often provide caching mechanisms to avoid using the database as much as possible, and they allow the programmer not to worry about the details in the database implementation, as this is handled by the ORM

tool. In many cases this could lead to less code in the data source layer. [**lessCode**] Examples are Hibernate [**hibernate**] for java, MyBatis[**mybatis**] for Microsoft .net and Java, and LINQ [**linq**] for Microsoft's .NET. It is important to point out that even though these frameworks hide the complexity of object-relational mapping, it does have some pitfalls. Many developers argue that by using ORM-tools you loose the ability to exploit the full features of a database management system. [**ormlame**]This includes the ability to do customized database tuning, and take advantage of special data types that are supported by specific vendors. Plus, the fact that an ORM-tool does indeed hide the object-relational mapping code, makes it hard to do debugging and write complex and efficient queries. At last, there might be some performance overhead provided by the code that is generated by the ORM implementation.

3.2.2 The Front-End

In the previous section we discussed the backend implementation of a traditional web app. The backend covers the majority of the application's source code, as it is responsible for almost everything, except displaying the HTML page. This is done on the client, and it is referred to as the front-end implementation. The front-end consists of the set of HTML pages, CSS style sheets and script files that makes up the user interface of the application. An important intent with the thin-client architecture, is that the client browser does not have to perform any heavy script computation or business logic operation, because everything is prepared by the backend.

Typically, when a client first accesses a given web app, the request goes to the server which responds with an HTML file. This file includes CSS stylesheets and JavaScript source files, either as references to external sources, or embedded inside the HTML. Also, the HTTP response usually includes a request for the browser to form a cookie with a given session identifier. The browser will immediately start rendering the HTML file. When the DOM tree is built and ready, the browser will generate a DOM-ready event. This event triggers the execution of some minor JavaScript initialization code. This code is responsible for setting up event listeners and connecting appropriate event handlers to them. When the whole page is fully loaded with all external sources fetched, the browser triggers the onLoad event, and the page is ready to be used.

Typical actions the user performs when navigating around on the page are either handled by HTML input forms or hyperlinks, or a JavaScript event handler that is registered with an event listener to execute an interactive request. If a user action is meant to result in a simple and interactive operation (like a displaying a pop-up window or an animation effect on a mouse-hover event), the event is usually handled by a JavaScript event handler which executes the operation quickly in the browser. Everything else is performed by an HTML form or a hyperlink. Every HTML form or hyperlink has a one-to-one mapping with a corresponding controller handler on the server. This mapping is represented in the form of a URL. When a form is executed or hyperlink is followed, an HTTP

request is sent to the server. The request holds parameters that are meant for the controller handler. These are received in the controller handler on the server as normal function parameters, that the controller will use to execute the action. The parameters were created in the HTML form, or in the hyperlinks, embedded in the URL itself. The controller handler on the server executes the action, and creates a new HTML page that is sent back to the browser. The browser will then again repeat the initial scenario where the new page goes through the rendering process, external sources are fetched, and new JavaScript event listeners are initialized.

3.2.3 Platform environment

Another aspect of the classical three-layered architecture is application tiering [2]. A web application is often divided into three tiers; the presentation tier, the logic tier, and the persistence tier. A benefit by having this separation of physical tiers, is that the tiers can be re-used and distributed in a parallel computing environment to gain performance benefits. However, it does bring communication overhead when data is transferred between tiers, plus that synchronization can become a challenge.

The web server hosts the presentation tier which communicates with client users through HTTP. The web server listens on port 80, which is the port number used for HTTP. The HTTP request is forwarded directly to the appropriate code on an application server. The application that runs on the application server communicates with the persistence tier that is usually hosted on one or more database servers. During the development process of an application, all the tiers are normally deployd on the same machine. This means that the developer has an app- and web server running on the localhost IP address on his personal development computer. In a bigger production environment, it is normal to distribute the tiers into separate physical server machines. Also, for large scale deployments, each tier can be distributed to multiple servers. In this case, a load balancer is used to delegate requests to available servers, and manage the communication. This is showed in figure 3.1 on the facing page. Each server is hosted on a separate machine in a so-called shared nothing manner, meaning the servers on each tier doesn't communicate with each other. This makes it easy to add more servers on demand without any synchronization difficulties. Note that when the persistence tier is composed of a relational database management system, a shared-nothing architecture is difficult to implement, due to the nature of the relational model [**dataCloud**]. This makes it difficult to do horizontal scaling with relational database systems. However, with applications that mainly performs read-tasks, one approach is to have a master-slave architecture, where one master node makes the

[2]Application layering is a term that divides the code into separate logical software layers. Application tiering on the other hand, is another logical separation often associated with where these "tiers" are physically deployed. For instance a three-tiered web application could have its presentation tier on the web server, logical tier on the application server, and persistence tier on a database server.

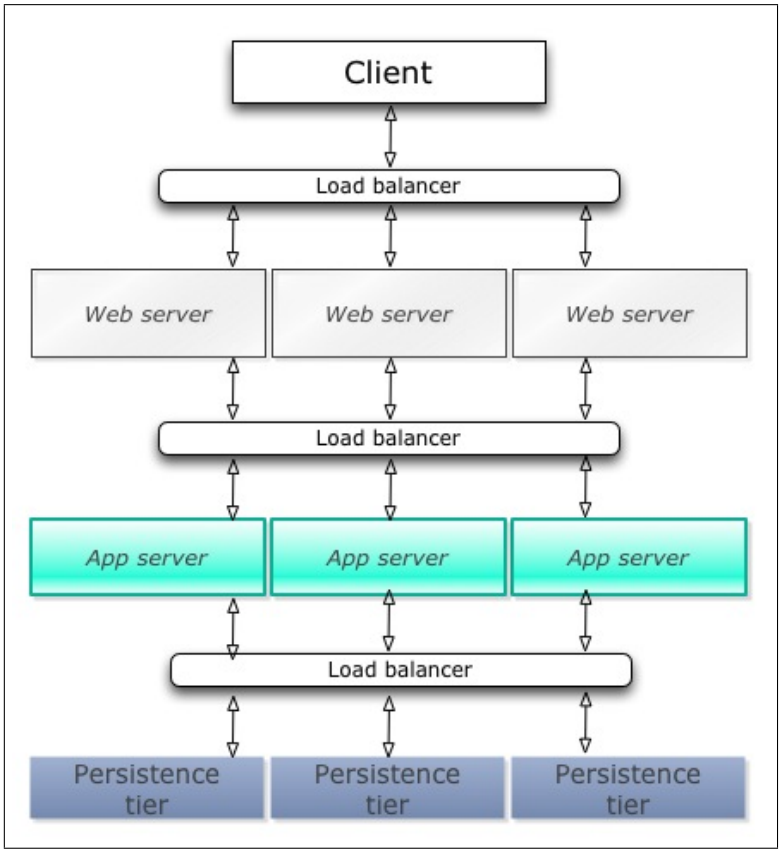

Figure 3.1: A distributed production architecture

writes, while other slave nodes offers read operations. Each time a master updates a table, the update is propagated to the slaves.

3.2.4 Examples of traditional web architectures

In this section we will look at two examples of some common, traditional three-layered Web application architectures. The examples use Web application frameworks that is made for different programming languages, and promotes different architectural solutions. The purpose is to propose a set of popular web architectures, that will be used as a base to determine the architecture we use to design the first prototype in this project.

MVC with Ruby on Rails . The Rails framework for Ruby has become a highly popular backend technology for web applications with big commercial users like Twitter and Git-Hub [**railsPop**]. The framework is built around the MVC design pattern [**mvc**]. In a Rails application, the controllers act as thin classes that receives a Url request, and delegates business logic to the Models. A Model is a Ruby class that implements the Domain Model design pattern, meaning it represents a particular domain resource in the application (for example a Movie, a User, a Book order

etc), and it has all the business logic operations that are relevant for that particular resource. The Model objects also knows how to do database mapping, by communicating with for instance a relational database. The view layer consists of template files written in a template language like HAML[**haml**] or ERB. These templates can reference data variables in a Model class, and are rendered into HTML files on the server before they are sent to the client's browser.

Ruby on Rails encourages developers to use RESTful URL's so that these are automatically mapped to specific controller actions. The Rails developer can define the set of (RESTful) URL's that the application will support, and the framework will in return automatically create controller handlers for each URL with appropriate model objects that are being referred to in the URL. This principle is called *convention over configuration*, because the framework sees the RESTful convention in the URL where a Model is referenced by a URI, and the operation is defined by either one of the HTTP methods GET, PUT, POST or DELETE. The controller actions usually calls a particular CRUD operation on a Model object that is referenced in the URL. This is again accomplished automatically in Rails by using the convention over configuration principle. This is applied to the data source layer, where database tables are automatically created given the structure of a Model class, meaning that table names and attributes are created by default, determined by the names and types in the Model's ruby class. For instance, if a Model class is named *Movie*, Rails will create a database table called *Movies* that has all the fields that exists in the Movie class.

The data source layer in Rails is typically built with the **Active Record design pattern**. This works by letting the Model objects implement CRUD operations for manipulating the database. This way, every Model class contains business logic operations (e.g validation, and business rules) and the functions requires to persist and manipulate the particular Model in the database.

The Rails framework makes it possible to build web applications quickly with a small amount of code, because Rails generates many of the basic structures and operations previously mentioned. In addition, the loose coupling that results from the MVC pattern makes the application easy to distribute in a cloud environment, because if designed properly, each tier has a shared-nothing relationship. They are independent, and can be cloned and deployed on separate servers, as depicted in figure 3.1 on the previous page. This is much of the reason that Rails has gained a lot of popularity lately in the Web application industry [**someREF**]

Front Controller with SpringMVC SpringMVC is a web framework that wraps the Java Servlet API technology. The Java Servlet API are a set of classes that implements low-level protocols to communicate on the web (e.g HTTP). For example, the Java Servlet API implements an interface called *javax.servlet.http.HttpSession*, which provides session management. This session implementation puts an Id (Called the JSESSIONID) in the

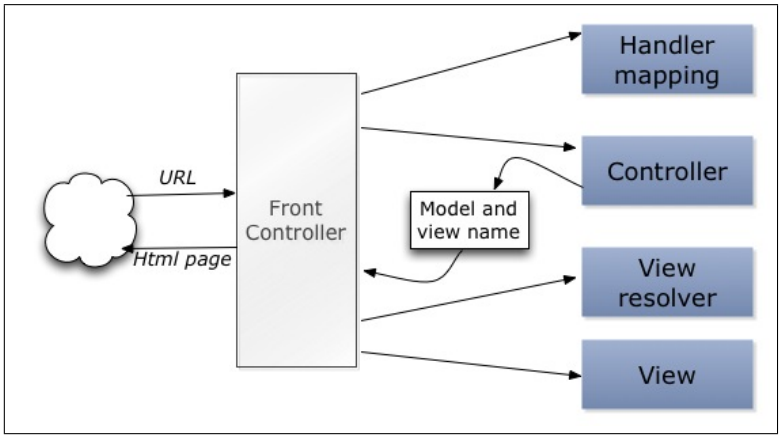

Figure 3.2: A request flow with spring MVC

User's Http cookie. This Id refers a particular HttpSession object that belongs to the User, and it is maintained by Spring on the server. If the User does not allow cookies, Spring will instead rewrite every Url in the App to contain the JSESSIONID. The HttpSession objects can also be set to have a time-out value, in order to avoid wasting too much memory on the server. Also, The HttpSession object maintains an object container, such that the developer of SpringMVC Web apps can use this container to store state information.

Just like with Rails, SpringMVC is also built around the MVC pattern. However, the framework also applies an architectural pattern called the Front Controller pattern. This pattern works by having one central servlet (the front controller) that receives all Http requests, and delegates control to an appropriate component that processes the request. This can be seen in figure 3.2.

When a request comes in, the front controller will ask a **handler mapper** to get a reference to a specific **controller** based on information provided in the request URL. A controller is a SpringMVC component (related to the Controller in the Model-View-Controller pattern) that is responsible for processing the actual requests. The handler mapper, is another Spring component that in addition to knowing what controller to issue based on a URL, performs pre- and post processing procedures, like validating the input from an HTML form, or marshaling an XML object to a Java class. When the handler mapper is finished pre-processing a URL request, it returns a reference to the controller that is meant to process the request. The front controller then delegates the request to this controller. The controller is usually responsible for performing some business operation implemented by the application programmer. When the business operation completes, the controller handler populates a *model* object with necessary data that is to be displayed in the **view**. The model object is a simple key-value data structure that lets the developer easily reference its data from the view file. The controller function also returns the name for the view that is to be rendered together with the Model object. The **view** is a template file

written in a template language like JSP or Velocity. It is the **view resolver** that maps a logical view name (e.g "homePageView") to a physical view name (e.g "/WEB-INF/views/homePageView.jsp"). Finally the view and the Model object is rendered to HTML and send back to the client.

With Spring comes a great implementation of an inversion of control container (IOC)[**ioc**]. Inversion Of Control is a programming methodology where the concrete types of object references are not know at compile time, because the references are instantiated and populated by an assembler at run time. This avoids having tight couplings between classes, and promotes a flexible codebase. The IOC container is a module that is responsible for creating objects, populate their references to other objects, and manage their complete lifecycle. The container is fully configurable, which makes it easy to decide how the classes are instantiated. For example, objects can be configured to be lazily instantiated, meaning the object won't be instantiated until it's first referenced. Also, there are multiple ways Spring's IOC container can be configured to create objects. In Spring, this is referred to as the object's **scope**. Some common scope-alternative are:

- Request - one instance is created for each HTTP request
- Session - one instance is created for each Http session
- Singleton - only one instance of the given class is created.

The IOC container provides a component based structure for the application by following the dependency inversion principle [**dip**]. This principle states that objects should not depend upon details, but rather abstractions. This means that the spring IOC container will inject concrete objects into components (which are interface references and not references to concrete classes) at run time. This enhances the decoupling of classes, because they have the freedom to decide the types of their owning references at run time, instead of compile time. The container works as an object factory, and makes it easy to add new components to the application while at the same time hiding the implementation details of the added behavior. An alternative way of composing the application with external modules is to use the service locator design pattern [**j2ee**], which is a common approach taken in Java EE based applications. Dhrubojoyoti Kayal [**Kayal**] shows how the service locator pattern can be implemented in Java Spring applications. However considering Spring already has an IOC implementation, it might lead to a lot of boilerplate code to assemble components using the Service locator pattern.

Another nice feature with Spring is that it has built in support for both relational- and noSQL database management implementations, as well as transaction management. On top of this it supports annotation-based declarations making it easy to customize features using java annotations [3]

[3]Annotations: A way to add meta-data information in the source code. Metadata is often provided by configuration files, but annotations enables this information to be populated directly on the java declarations in the class files. An example is to use annotations to define how each variable in a java class is represented as a SQL column.

This conforms to the convention over configuration principle, which leads to simple and quick web application development.

A typical domain logic layer in a Spring application is built with the service layer pattern [**serviceLayer**]. In the service layer pattern, the business logic is split into two. A service layer that exposes an API that encapsulates all the business operations, and the domain model which structures the application's domain into nouns, typically represented as separate classes. The API, or service layer is categorized into logical abstractions called **services**, where each abstraction is hidden behind a facade [**facade**]. A facade is an interface that contains simple access methods to a more complex set of data structures, like complex business operations or database access methods. Hence, each service encapsulates complex business operations and communicates with lower layer data source functions. The domain objects typically has no logical functions, only private data attributes with accessor methods. The service classes communicates with the data source layer. The service classes are also responsible for handling transaction management, so that if a transaction fails, the service classes knows how to handle it. When performing transactions, the service classes delegates to the data source layer which would know how to communicate with the database and perform object-relational mapping. This could be done by implementing a customized object-relation mapping scheme, for instance by implementing the data mapper pattern described earlier, or by using an ORM tool like Hibernate or JPA. The latter case is similar to the active record pattern used in Ruby on Rails. Spring is one of the most popular frameworks for Java, as of 27th February 2013.[**springPopular**]

TODO: Discuss front-controller vs MVC vs other shiet like page controller in php, template view.

Other examples Another popular architecture for thick-client Web apps is the **Page controller** pattern which is a commonly used with the Sinatra Web framework for Ruby. In this pattern, there is a strict relationship between a web page and a controller. Every (HTML) page on the app has a dedicated controller, such that the controller knows how to handle requests for the page it is responsible for, and it knows how to generate that page given the result of the request that is performed on that page.

3.3 Modern Web Application Architecture

The motivation for proposing a reference model for modern Web app architecture has its roots in an architectural shift that started around 2008-2009.[**hales64**] At this time, a new form of client-server interatcion scheme was emerging away from the traditional hyperlink/form-oriented request-response scheme, where each time the client interacts with the application, a request is sent to the server in which a new HTML page is created and sent back, and the browser has to reload the entire page. In addition, AJAX technology had been around for a long time, which usually was being

used when some data is needed from the server for some minor interactive behavior. Building applications completely based on JavaScrip with Ajax for server communication was possible, however most developers had acknowledged the fact that such front-end oriented architectures doesn't scale in terms of code flexibility; it often ends up as piles of spaghetti code. This is partly because of web developers' relationship to JavaScript; as stated earlier, JavaScript has had its buggy features and lots of browser incompatibility problems since its early beginnings. Also, many professional developers has a misunderstood relationship to the language; they are not aware of JavaScript's key language features like object-orientation including prototypal inheritance, and functional- and dynamic programming facilities, having closures and a dynamic typing system. Also, developing large-scale JavaScript applications is difficult, because the language lacks constructs that facilitates flexible and modular code. It is possible to create namespaces and class-based structures with JavaScript, but it requires a lot of effort because it needs to be built from the ground up in every application. However, this was only until recent years, because a lot of work has been done to build frameworks that neatly solves these problems.

3.3.1 Motivation

The architectural shift began in 2010, when browser's capabilities to execute JavaScript increased tremendously. This was by the time when Google launched its Chrome browser with the powerful V8 JavaScript engine, and the compelling browsers followed along with similar JavaScript capabilities. Also, The JavaScript language itself has started to get much more endorsement from the web community with the standardization of ECMAScript, and Google's provenly working large-scale JavaScript applications like Gmail, Google Maps and also Node. This has led to many new experimental web architectures that takes a distance from the thick server model, and where application state is gradually moving from the server and into the client. However, not only is the application state moved to client, but also the application logic. This means that the core application moves from the backend to the client, leaving the backend being a simple and agnostic storage center for persisting the domain data in the application. Not only is this feasibly, but it's becoming increasingly popular.

3.3.2 The Idea

Handling state on the client means that the one-to-one mappings between user interactions and controller handlers on the server are gone. Form requests and hyperlinks aren't sent directly to the backend, but instead handled on the front end. The client only contacts the server when domain data is needed to be fetched or manipulated in the database, authentication of users, or the browser needs to fetch static content like for instance images, movies, stylesheets or script files. In Reference Model 2.0, we

propose an architecture where most of the responsibilities in a Web app is moved from the server to the client. These responsibilities include:

- Routing between pages
- Render views into HTML
- Sessions and state handling
- Business logic operations
- Input validation
- Language translation of content
- Deciding what to store in the database and when

Now, there are variations in how many of these responsibilities that are performed mainly by the client. Input validation for instance usually needs to be done on the server in addition to client-side validation, in case the user manages to contact the server directly without going through the front-end. Also, some applications might prefer to let the server be involved in routing between pages [**airbnbnode**], and perform some of the page rendering (ref to twitter). There are many good reasons for moving the backend responsibility to the client's front end. Some key advantages are:

- Centralization of the application's codebase, since much of the application logic, and the dynamic presentation logic is now located in the same place, and implemented in the same programming language
- Less load on the server because more processing is moved to the client
- Facilitates smart front-end techniques like lazy loading of HTML- and script files, because JavaScript techniques can be used to only fetch HTML and script files when needed.
- The presentation logic and application logic lies much closer together in terms of source code. This facilitate tight cooperation and integration and be built with familiar design patterns in only one language: JavaScript. Such tight cooperation where hard to implement with the traditional approach, because it required tight cooperation between a template language (e.g JSP), JavaScript, and a backend language like Java.
- Enables the server to offer a more general-purpose interface to the outside world. It no longed serves to deliver content to one particular client, but also other types of clients like third-party applications

3.3.3 The Backend

The responsibility of the backend is mainly to manage the database. It's interface is still exposed as controller handlers, however these are not customized for particular HTML form requests or hyperlinks that leads to a new HTML page. Instead, the backend exposes an **API** for interacting with the application's domain data. This API contains a set of public functions where each function is identified by a specific URL. Each URL refers to a domain entity in the application, and an operation that the server is to perform on the domain object. Note that this operation is usually not a complex business operation, but merely a single database operation. The operations offered by the backend API are commonly expressed using merely HTTP methods. Hence, the server API is a RESTfull service that adheres to the principles in the REST design pattern. Now, instead of creating and returning a complete HTML page upon each client request, the server would return a more fine-grained data object represented in a uniform data format like XML or JSON. This is a much more general-purpose solution, because external clients like mobile applications and other third party applications can now use the service offered by the application, because the service will no longer only respond with entire HTML pages, but a fine-grained format that is easier to manipulate and use as desired. This is often referred to as a service-oriented architecture [4].

3.3.4 Front-end Frameworks

Another interesting aspect of modern web application development is the evolution of JavaScript development environments. Not only has JavaScript been judged for being a language with many limitations, but it has also lacked proper development tools like editors and debuggers, and frameworks for syntactic sugaring. Even though JavaScript is, and has always been a highly popular language for web applications, it has mostly been used as a tool to add dynamic behavior, input validation and simple Ajax calls to the server. With the increasing interest for large-scale JavaScript client applications, a huge amount of frameworks and language variations have been built. This includes:

- A huge amount of frameworks for structuring and organizing JavaScript code. [5]

- Programming languages that compiles to JavaScript, to facilitate the development of large-scale JavaScript applications with a language that is more similar to traditional languages like Java or Ruby. Examples are Coffescript (ref) and clojure script(ref).

[4] Many service-oriented architecture experts does not consider REST as a part of the SOA-paradigm, mainly because it is resource-oriented rather then service oriented

[5] There is an project (REF) on www.Github.com(REF) where open source developers are implementing the same web application with different JavaScript frameworks to help developers choose a proper code organization framework for their web apps. (REF)

- Syntactic sugaring frameworks that provides utility functions for operating on various JavaScript data structures, doing mathematical operations, or fixing browser compatibility issues.
- Rendering engines for the front end, built to render HTML code with data objects.

3.3.5 The Database

Together with this thick-client approach one has also seen a sudden interest in alternatives to the traditional relational database. With the increasing popularity of applications being deployed and run in the cloud, there is a need to be able to distribute an application's database over many servers. Now it so happens that normal relational database structures has showed to have transactional challenges, and lacks scalability when distributed in the cloud [**dataCloud**]. These issues, together with the demand for extreme high performance, has led to the development of new persistence technologies that are specifically designed to be replicated over multiple distributed database servers. The so-called noSQL paradigm is a common term for classification of database technologies that does not belong to the family of relational databases. In addition to being optimized for distribution, many noSQL database have addressed a flexible schema structure, or even not using schema's at all. This is in favor for the ability to customize entities in highly flexible manners, that would suit the application, unlike SQL which favors schema design that fits the entities and their relations.

Many noSql database have gained massive popularity in the last couple of years, because they has showed to deliver excellent performance and scalability, and because they sometimes fit more to the application's data needs.

In the next few sections, I will dig further into the ideas and technologies that typically makes up the modern web-app architecture.

3.3.6 Thick client tier

In the traditional architecture, almost all of the business logic is implemented on the server, and the only code that is executed on the client is the JavaScript functions necessary for generating dynamic behavior in the browser. The idea in this proposition however, is to move the application's logic to the front-end. The business logic involves the domain objects and the operations that are performed on them, data validation, language translation of data, and communication with third-party API's. Instead of programming this logic in a proper backend language like Java or Ruby, this is implemented in JavaScript, or a language that compiles to JavaScript, like CoffeeScript or Clojure script. The front-end source code is located on the web server as publicly available static assets. All the source code has its root in the main HTML file which is sent the client the first time he uses the app. The HTML file contains references to the JavaScript source file(s)

that makes up the front-end application, such that the browser automatically fetches these when the main HTML page is being rendered. Typically the JavaScript code is separated over multiple source files. Also, the HTML page will have references to external JavaScript framework files that are not developed by the application programmer, but is used inside the app's own JavaScript code.

The JavaScript code can either be sent to the client all at once when the web application is first accessed, or it can be lazily fetched. The latter means that the client will only ask for the necessary part of the code, when it needs it. This requires the source code to be split into separate code files so they can be sent individually from the server. This might be a performance benefit in case the whole JavaScript codebase is very big. A pitfall however might be that very many small JavaScript files are required at the same time. In some cases the TCP connection overhead might lead to a performance bottleneck because the browser usually creates a new TCP connection every time the browser requests something from the server. Sending all the scripts (and HTML and CSS files) on startup gives the advantage of being able to gather all the JavaScript logic into one single file that can be extensively minimized and compressed. In this case, no further source files are needed to be fetched from the server, but the initial load time might be overwhelming. It is also possible to do a compromise, where the programmer gathers the most commonly used scripts into one compressed file that is sent initially, and then lets the client fetch the rest only when it is first needed.

3.3.7 Single-page Web Application Architecture

One essential advantage of the thick client architecture is that server requests can now be limited. In the traditional approach, each user interaction with the page leads to a server requests that result in a new page, and the browser has to reload the whole page. This causes a disruption in the user experience. With the modern approach, the request goes straight to JavaScript event handlers. This way, the client stays on the same page during the whole session, requiring no new page reloads. If for instance a link in the navigation bar that leads to a different "section" in the app is requested, everything is done in the browser by manipulating the DOM tree so that the page moves to a new state. This leads could to a much more fluid user experience, because server requests can be avoided, and the browser does not have to reload the entire page. This principle is commonly referred to as a Single-page app (REF).

If the front-end needs to synchronize data with the database it will send an asynchronous to the server. This could for instance be to save some data, or get some new data that needs to be displayed on the page. The client can also save data in the browser's memory, such that the more domain objects stored in the browser's JavaScript memory heap, the less requests has to be sent to the server. Depending on the application, write, update and delete operations will always sooner or later have to lead to a server request. In applications that requires all updated data

to be available as close to real-time as possible, the operations has to be directly written to the database, in effect work as a write-through cache. In applications where this requirement is more relaxed, the front-end can choose to perform the server persistence at a later, more appropriate time. For web 2.0 applications, the former is often wanted, because users usually want to be see the latest updated data at all time.

3.3.8 REST API's and JSON

The thick client model avoids letting the user communicate synchronously with the server. Instead, the JavaScript application that runs in the browser is responsible for knowing when it needs to communicate with the server. This would be whenever some domain objects that are not already in the browser's heap are requested, or some domain object must be persisted to the database. The requests to the server are exclusively done through a RESTful API. This means that all domain objects that are to be offered by the server, must be accessed through one of the HTTP methods *GET, POST, PUT, or DELETE*. An example of a RESTful API that offers functions for persisting a Shredder object is showed in table below. A shredder is a guitarist in the web-app prototype that has been created in this thesis.

Method	**URL**	**Description**
Get	www.shredhub.com/ shredder/1234	Get shredder with id 1234
Post	www.shredhub.com/ shredder/ ?name=Jude Swayer	Add shredder with name Jude Swayer
Put	www.shredhub.com/ shredder/ 1234?country=Sweden	Update shredder with id = 1234 set country = Sweden
Delete	www.shredhub.com/ shredder/1234	Delete shredder with id 1234

The server would respond with the domain objects in JSON format, instead of a complete HTML file. In effect, the return value is much more fine-grained. Also, the API is very consistent, because it adheres to a common interaction scheme, namely the HTTP functions Get, Post Put, and Delete. This creates a familiar and easily-to-understand server API. Many modern web application frameworks like Ruby on Rails, Spring MVC and Django are built based on the principles of REST, and will automatically create RESTfull controller functions based on the application's domain objects. This programming interface works really well with the thick client model, because now the client tier can be 100% responsible of maintaining the application's state, and use the backend as a simple persistence API to store and deliver the domain objects. Hence, the backend is just a simple service that knows nothing about how to visualize the domain objects in HTML, leaving this responsible to the client tier.

Modern REST API's very often use JSON as the transmission format. The reason is that it fits well into the programming model both on the

front end and backend, because considering that the front end code is implemented in JavaScript, and JSON is part of the JavaScript language, it is very appropriate to use JSON as a transmission format because no marshaling has to be done on the client. At the same time on the backend, it is so that some of the noSQL technologies uses JSON objects, or JSON-similar syntax for persisting data. Hence we get a common transmission format that can be used across the whole software layer stack.

3.3.9 Modular JavaScript

TODO: rewrite this:"" How are pieces of JavaScript code defined today? Defined via an immediately executed factory function. References to dependencies are done via global variable names that were loaded via an HTML script tag. The dependencies are very weakly stated: the developer needs to know the right dependency order. For instance, The file containing Backbone cannot come before the jQuery tag. It requires extra tooling to substitute a set of script tags into one tag for optimized deployment. This can be difficult to manage on large projects, particularly as scripts start to have many dependencies in a way that may overlap and nest. Hand-writing script tags is not very scalable, and it leaves out the capability to load scripts on demand.""

A modular codebase is made up of highly decoupled, encapsulated pieces of coherent features that are implemented in separate modules. A codebase that consists of loosely coupled modules, facilities a flexible and maintainable system, because the codebase contains less dependencies[**henrik**]. This makes it easier to change one part of the system without harming any other. As previously stated, the JavaScript programming language does not have module features built into the language. This means that it is up to the developers themselves to develop some sort of module framework. Various design patterns have been proposed to establish standard ways of developing modules, like the module and sandbox pattern [**jspatterns**]. A lot of work has been done to provide open solutions for JavaScript developers to build modular JavaScript code, the two most famous being AMD (Asynchronous Module Definition) and CommonJS. Having the JavaScript code separated into modules means that these modules can be split into separate source files. That is what facilitates the lazy loading of JavaScript files previously mentioned. The modules can even depend on HTML files, so that whenever a JavaScript module is loaded, the HTML page will be loaded as well, and will be available as a text string inside the JavaScript module. With this feature, we can now state that the JavaScript modules are the first-class citizens in the application and hence decide which and when a given HTML page is to be displayed in the browser. This is an important difference from Reference-model 1.0 where the HTML pages where the first-class citizens, and the JavaScript code was just embedded inside the HTML.

The AMD principle was made to have a better alternative to loading scripts then the traditional group of <*script*> tags embedded in HTML files. AMD brings an API that facilitates the separation of JavaScript into

modules and defines the modules' dependencies to other modules. These dependencies are asynchronously loaded into the module, which avoids browsers having to block while waiting for synchronous module loading. the AMD API comes with two functions: require() and define(). Define() is used to encapsulate a JavaScript module while at the same time define the other module it depends on. The require() is used to asynchronously load modules into a function, in which the function will not be called until all the modules are loaded and ready to be used inside the function.

3.3.10 Client side page-rendering

In Reference-model 1.0, every HTML page is completely rendered on the server and never changed after the page is sent to the clients. In Reference-model 2.0 however, this idea is completely abandoned. HTML pages are rendered "on the fly" with JavaScript code that is executed on the client side. The HTML pages contains templating markup that is detected by a JavaScript rendering engine. Whenever a new HTML page is to be loaded, or embedded into the current HTML page, a certain JavaScript function will be called. This function is responsible for asking the rendering engine to take an HTML page together with a set of JavaScript objects that represents the data that is to be displayed in the page, and finally return the HTML page with the JavaScript data rendered inside it. As previously mentioned, the AMD model makes is possible to have JavaScript modules that depends on HTML files. This facilitates a nice programming model, because if the HTML pages are also separated into small, independent modules, then these modules can be stitched together to form complete HTML pages. For example, a JavaScript module might depend on a small HTML module, and when the JavaScript module is loaded, it can render the HTML module and insert it in the current HTML page. As such, HTML modules can be reused, removed or swapped out from the current HTML page by the JavaScript renderer. This enables a highly flexible way of altering contents of the HTML page, and also to easily glue together reusable front end solutions.

3.3.11 Client state

Another part of Reference-model 2.0 is how the state is being kept between requests. The major goal of Reference-model 2.0 is to move much of the application logic from the server to the client. Hence, being able to keep the state client side is of high priority. In this architecture, we propose a solution to this by using HTML 5's Web Storage. The HTML 5 web storage is a standardization made by W3C that defines how to store structured data in the browser between page requests. It is supported in all modern browsers. HTML 5 web storage contains two storage containers: **localStorage** and **sessionStorage**. The difference is that local storage is being persisted even when the browser is closed, and it has no expiration date. The session storage is only kept in the browsers memory until the session is over, which means either if the user closes the tab or the browser.

The storage enables developers to store lots of more data then what it supported with cookies. As an example, Internet Explorer 8 allows for sessionStorage up to 10 mega bytes, while a cookie is limited to 4 kilo bytes. The sessionSstorage is consists of a key-value data structure that is accessed by a simple JavaScript API.

The session implementation is built by letting a JavaScript object be created when the web application is first accessed by the client user. The object is populated with account information for the user, and can be extended with data values that is appropriate to be cached in the browser. Each time the client tier changes state or receives some state information from the server, it can be persisted in the session object. Hence the server does not have to maintain a session object in memory for each user that is currently logged in to the web application.

3.3.12 Node Js

Node Js is a Web application framework built for using JavaScript as the programming language. This is one of the first (and most popular) solutions for developing JavaScript Web applications on the server. It runs in Google's V8 JavaScript engine. A big advantage this framework has compared to other frameworks like Rails and Spring, is that it is event-driven. This is a completely asynchronous programming environment that is centered around events, where clients subscribe to events are notified when the events are triggered. This avoids the blocking scenario that might occur in regular synchronous systems. It is completely possible use Node Js as the web application framework for Reference Model 1.0, the reason I mention it here is; because the backend is built with JavaScript when choosing Node, the application as a whole fits more with the Reference Model 2.0 which uses JavaScript as the core language on the front end. Hence you get a complete and pure JavaScript developer environment, in effect making the application more coherent, and less separated into different parts (backend/front-end).

3.4 The Solutions

The two reference models just described are popular approaches to how developers design and implement modern, interactive web apps. In this thesis, our goal is to compare these two approaches, in order to find any potential drawbacks or especial advantages. Therefore, we have taken the main ideas these reference models and applied them in two different software architectures for a prototypical web app. The first solution is called Architecture 1.0, while the latter is called Architecture 2.0.

Architecture 1.0 is a thin client/thick server application. The backend is centered around a three-layered software architecture. This means that all the view logic happens in the presentation layer on the server, business logic operations happens in the domain logic layer, state is being kept on the server using Http sessions, and the front-end tier is tightly coupled to

the server such that each Html form or link has a corresponding handler on the server which serves to generate a new Html page given the result of the request. The backend is build with Spring MVC, and is therefore a Java Web app. It uses a SQL database to persist data.

Architecture 2.0, is a thick client/thin server architecture where control resides on the front end. State management, controller handling and business logic all happens on the front-end, which is built purely with JavaScript. The front-end is built with a sibling of the Model-View-Controller pattern, where the models are active record objects that uses the backend-server only as a simple data repository. The backend is offering its services through a Rest API, which is built with Node js. The Rest API manipulates the database, which is built with MongoDB.

3.5 Summary

TODO: Rewrite this In this chapter we have proposed two very different web architectures. A short comparison of these two can be seen in the table below. The table sums up the major differences between the two architectures, where each row is concerning similar application issues.

Reference-model 1.0	**Reference-model 2.0**
Server-side page rendering	Client-side page rendering
Application logic runs on server (thick server)	Application logic runs in browser (thick client)
Session state stored on server	Session state stored in browser
Form-based interaction with complete html pages returned	RESTfull ajax requests for JSON objects and asynchronous module loading
Relational database system	Document-oriented database based on key-value semantics

Reference-model 1.0 had a thin client model with all the business logic performed on the server. The server's job was to perform the business operations, execute database operations and create and render a view that is sent back to the client. I argued that having a decoupled three-tier architecture could make it possible to distribute each tier in the cloud, but the problem with having a relational database could lead to that being a bottleneck. The latter approach had a thick client model where most of the logic was performed in the client's browser (I.e front-end), primarily using the server for database operations. This could lead to a structured and loosely coupled front-end implementation, as well as a reduced amount of data sent back and forth between the client and server. The RESTfull architecture, together with asynchronous HTML/JavaScript loading can make it possible to limit the data that is sent back back from and to the server, since the only time a complete Html page request is required is the first time the page is accessed. However the first HTTP request might turn in to being a very big data package, since a lot of JavaScript has

to be sent to the client. This could in worst case could lead to a very slow initialization time. Also because the data format is cross-platform, other external clients like third party users and mobile applications can use the service offered by the application. Finally, the noSQL database characteristics seems to be more suitable in a cloud environment, because it is possible to shard the database tier with in a shared-nothing manner. It also provides programmer friendliness due to its simple syntax. At the end of the chapter we stated that the two reference models discussed in this thesis are used as a base for designing and implementing the two architectures that have been built for this thesis.

Part II

The project

3.6 Overview

This part covers the project that has been developed for this thesis. It contains a Web application that has been built twice with two completely different architectural approaches. The main goal in this thesis is to find a superior software architecture for a typical Web 2.0 application, by comparing a traditional architectural approach with a modern and innovative approach. In order to evaluate these two architectures, the writer of this thesis has invented a Web app that represents a traditional Web 2.0 application. This Web app contains the most commonly seen features of a traditional web 2.0 application. This includes:

- Social networking interactions:
  - Have a user-profile that is publicly visible to other users
  - Connect to other users, for example in a friendship relationship
  - Creating blogs and upload posts to it
  - Ability to comment and rate blog posts
- Interactive behavior with rich user interfaces
- Large amounts of persisted data (this mainly because the app-users themselves create the information content)

The first architecture that was built conforms to a traditional approach, and the second conforms to a modern approach.

The rest of this part is separated into three chapters: In the first chapter we will look at the Web app itself, seen from the end-user's perspective. A description of the Web app's user behavior and requirements is necessary in order to understand the solutions that was taken when designing the software architecture for the app. The Web app is named Shredhub. In the last two chapters, we will have a detailed look at the two ways the app was built. The architecture outlined in the second chapter, is named Architecture 1.0, and is adheres to the principles from *reference-model 1.0*. The final chapter discusses the architectural details in Architecture 2.0, which is based on *reference-model 2.0*. For the record, during the discussion we will use the term *User* to refer to the currently logged in user.

Chapter 4

Shredhub, a Web 2.0 Application

4.1 Shredhub

Shredhub, is a social web application for musicians, aimed primarily for guitarists. The application enables users to share their skills and musical passion in a social and competing manner. Through a modern and interactive user interface, the users are able to post videos of themselves playing a short tune. Everyone can watch, comment and give a numbered rating to the videos, so that the creator can achieve experience points and become highly ranked on this social platform. The purpose of Shredhub is to gather guitar players from across the world to create a community of competent and enthusiastic musicians. Having a social network of people who share the same interest is a great way for people to both learn, and have fun with their playing. People can deploy their musical ideas, show the world how beautiful their guitar sounds, challenge a friend or stranger in a battle, or whatever would suit their needs.

Tt is important to acknowledge the fact that this application is primarily for guitarists, which does imply a slightly small user group. A better solution would be to implement a system that supported more kinds of musicians, for instance drummers, piano players, saxophone players etc. Therefore, a better solution could be to let the user pick their preferred instrument before they access the application's main page. From there on they would only be able to participate with the kinds of musicians the user picked at startup. However, because I have only had a certain amount of time to implement application, extending the application unfortunately goes out of project scope. Therefore I content myself with only supporting guitar players in this project.

4.2 User functionality

The term **shredding** has a broad sense in the context of guitar playing. Generally, it refers to a particular playing style that incorporates advanced techniques and fast playing. It originates from the field of rock music, but

many styles use the term shredding to refer to a particular quick melody played by a guitarist. In my application, I use the term **Shred** to refer to a video of someone playing a short guitar melody that uses some fancy technique. A tradition in the guitar playing community is to show off their "shredding" skills in videos on the Internet, like for instance Youtube or MySpace. Another fascinating scenario is the so-called guitar battle, in which one guitarist would battle against another on stage in front of an audience. In this scene two or more guitarists would take turns in playing shreds against each other, with the intent that each shred is more interesting then the one performed before. I have adopted these scenarios in the web application. Here, I use the terms **Shredder**, meaning a user on the web site. Users can register a (Shredder) profile on the page, where they upload some information about themselves, for instance where they come from, what sorts of musical instruments they own, or what types of music they like. A shredder has a shred-level that represents his shredding skills. Shredders are connected to each other as **Fans** and **Fanees**; a shredder A can be a fan of shredder B, in which case shredder A is a fanee of shredder B. This creates a graph of interconnected users, which is one of the most important requirements for social networking apps. A *Shred* is the name of a video that a Shredder uploads, and *Battle* is a competition between two Shredders. Everyone can give a rating to the shreds that are uploaded, and Shredders are able to comment on a particular Shred. A Shred will also have a set of **Tags** [1] assigned to it. These can be attributes like "riff", "solo", "song-cover", "rock", "classical", "sweeping", "tapping", "scale", "whammy-tricks" etc. Shredders achieve experience points when someone rates their shreds, either in a battle, or a normal shred upload. Experience points helps the shredder gain **level**, which is a measure of how skilled the player is. A major goal for the application is to create good personal shred-recommendations for each Shredder, so that they can discover new Shredders and widen their fan graph. The recommendations are created based on the Shredder's profile, such that each Shredder get recommendations for other, similar Shredders.

4.3 User stories

Given below is the set of user stories that has been implemented for the application. Each user story is outlined together with the URL that contains the user story.

The front page, www.shredhub.com

The user is first met with a front page as seen in figure 4.1 on the next page. Here the user can either register as a new Shredder, or log in with a username and password. The user will not be able to access any of the other services in the app before he is logged in. The page also displays a set of the most popular Shred videos.

[1] Tags is a widely adopted term in the world of Web 2.0; many web 2.0 applications uses tags to classify things like blogs and images

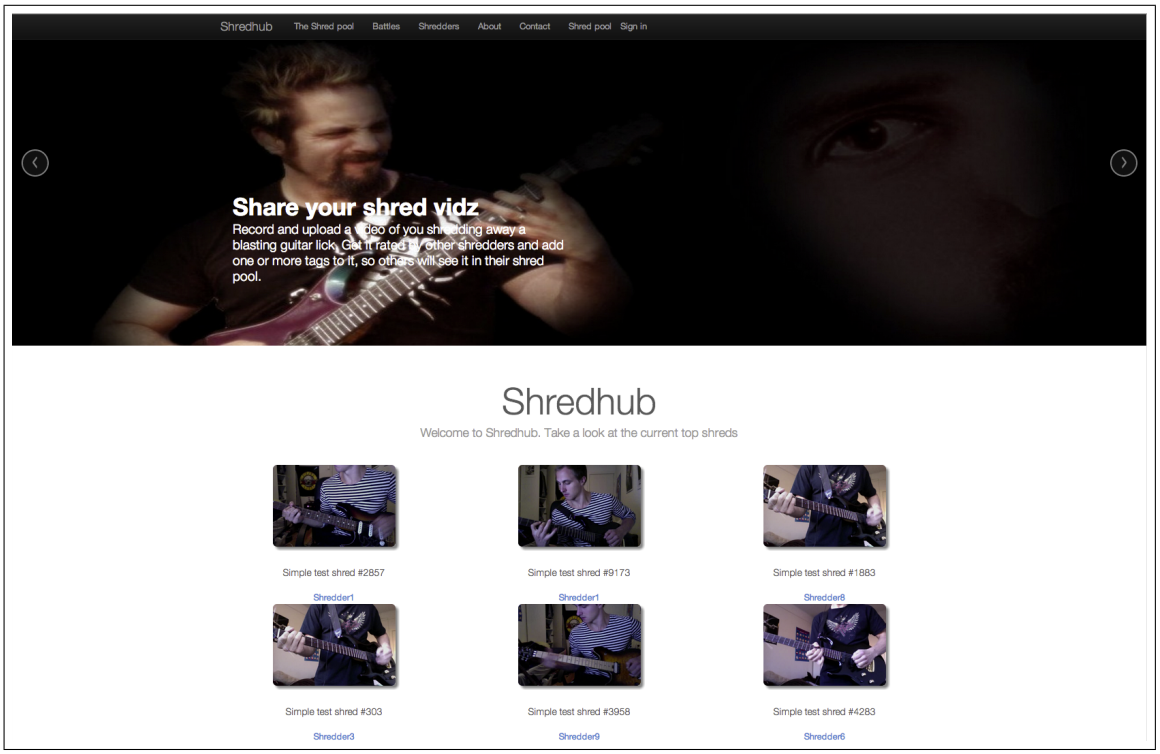

Figure 4.1: The front page at Shredhub

The shred pool, www.shredhub.com/theshredpool

This is the first page the Shredder meets when he logs in. It is the main shred-arena, which contains multiple rows of Shreds made by other Shredders. Considering it is sort of a central page in the app, it is important that the page renders very quickly. The page contains the following rows of Shreds:

1. Newest Shreds made in the app
2. Shred-news:
   (a) Newest Shreds made by fanees
   (b) Newest Shreds by fanees made in a Battle
   (c) Newly created Battles by fanees
   (d) New recommended Shredders to connect to
3. Shreds with particular high rating
4. Shreds from Shredders that might be of interest
5. Shreds based on tags the User enters

Every row contains a collection of 3-5 Shreds, except the final row which contains 20 Shreds. The User can click the next button in a row, which results in a new row of Shreds. This should also happen very fast and responsive. The Shred-newst section is a set of Shreds and Shredders especially picked out to fit the User's profile,

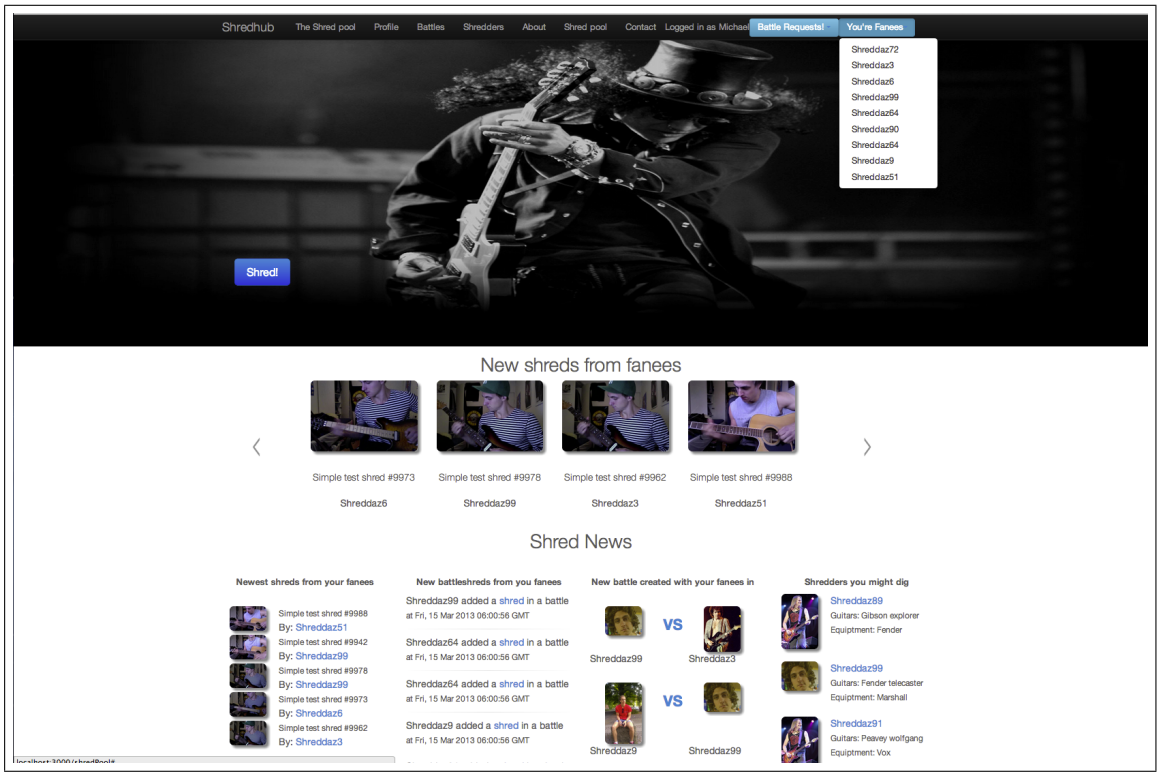

Figure 4.2: The Shredpool (top)

that is, new content made by his fanees, and recommendations for new Shredders. The recommendations are made to motivate the users to expand their network off Shredder-connections, which is an elementary part of typical Web 2.0 applications.

The shredder can also create and upload a new Shred by clicking "Upload shred". If this button is clicked the Shredder will be asked to pick a movie from his computer (or smart phone/tablet), add some relevant tags for the Shred, a description and a name. Then the Shred will be saved in the database, and immediately be available to every other Shredder in the application. This screen is showed in figure 4.3 on the facing page. If the Shredder clicks on a particular Shred one of the rows, a new window pops up displaying the Shred video, a description for the Shred, a name, and the set of tags created for the Shred. In addition there is a list of comments that has been made on the Shred and a number representing the computed rating for the Shred. The logged in user can choose to add a comment on the Shred, and give the Shred a rating value. Its very important that when the User adds a rating or enters a comment, the result of this is displayed very fast in the Shred window. If a rating is added, the new rating is added to the Shred, and the Shredder who made the Shred will earn more experience points.

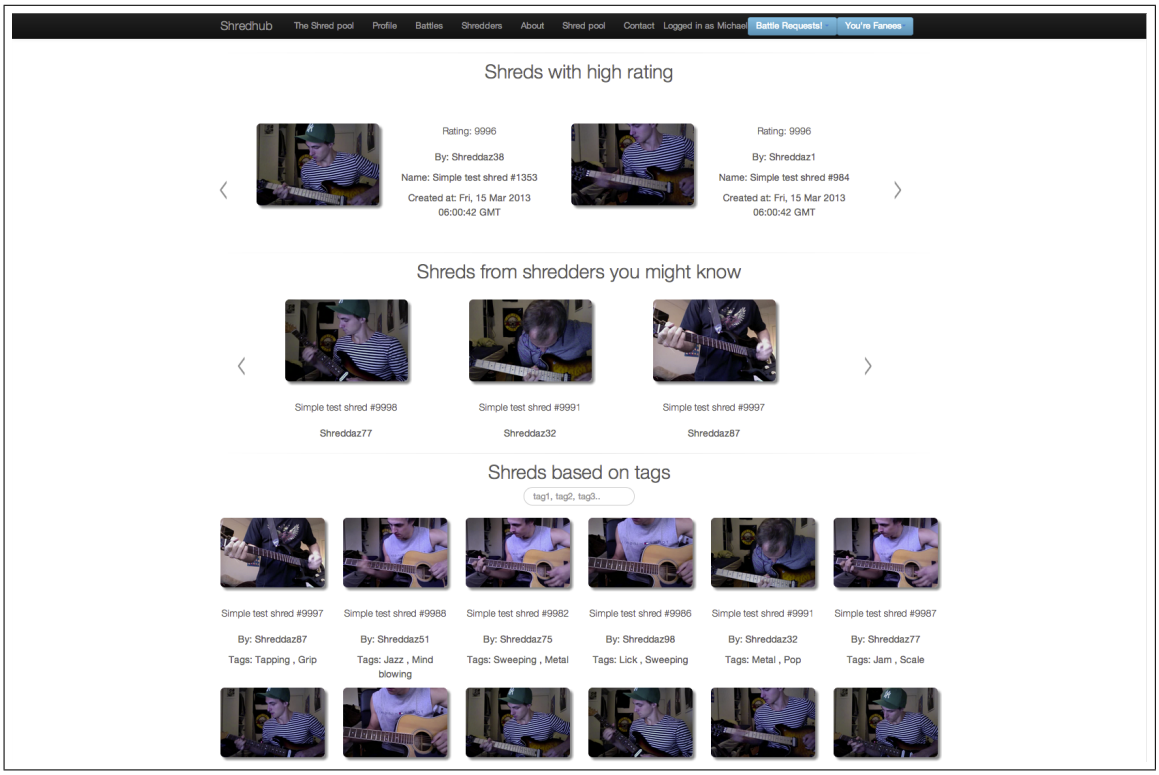

Figure 4.3: The Shredpool (bottom)

Shredders, www.shredhub.com/shredders

This is an overview of all the the Shredders that are using the app. Considering that the amount of Shredders on the page might be very big, the list is paginated, meaning a fixed number (20 in this case) is displayed at a time, and the Shredder can click next to iterate to the next page of Shredders. Shredders can also search for other Shredders by name. The purpose of this page is to encourage Shredders to meet new Shredders so that their fan graphs can be extended. The currently logged in Shredder can click "become fane" to become a fan of a Shredder that is in this list, or click on one of the Shredders to access his public profile page. This page can be seen in figure 4.4 on the next page.

Shredder, www.shredhub.com/shredder/<id>

This is a page that displays the details for a given Shredder, that has the unique id found in the URL. A list of Shreds that the current shredder has published is displayed in a list view, together with a list of his fans. The logged in user may choose to challenge this Shredder for a battle. The page is customized to show the relationship the currently logged in shredder has with this Shredder. This might be that they already are in a battle, or if a battle request is sent to this shredder, if they are fans of each other already, and other similar relationships (4.5 on page 71).

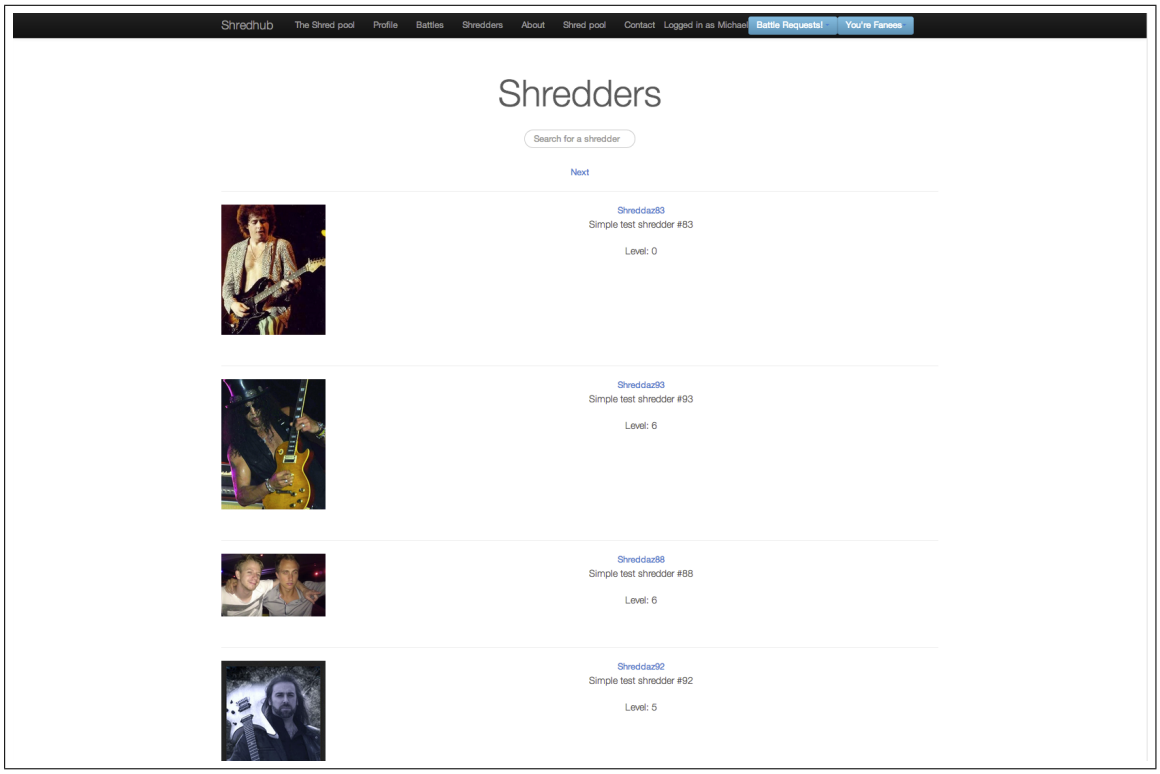

Figure 4.4: The list of Shredders

Battle

This page displays a battle between two shredders. If the currently logged in user is one of the battlers, the user is able to upload a shred for the battle. In a battle I distinguish between the battler who initiates the battle, and the battlee, being the one who is challenged. I have not added an image of a battle, because it won't be discussed in much detail in this thesis.

Shred

A pop-up window displays a particular Shred made by a Shredder. Users can add a rating to the Shred, and add comments for it. The shred can be accessed from multiple different pages in the app. An example image is given in 4.6 on page 72.

Upload a Shred

For uploading Shreds, a simple pop-up window is displayed so the User can add a Video, a description, and a set of tags. This window can only be accessed inside the Shredpool. This can be seen in **??** on page ??

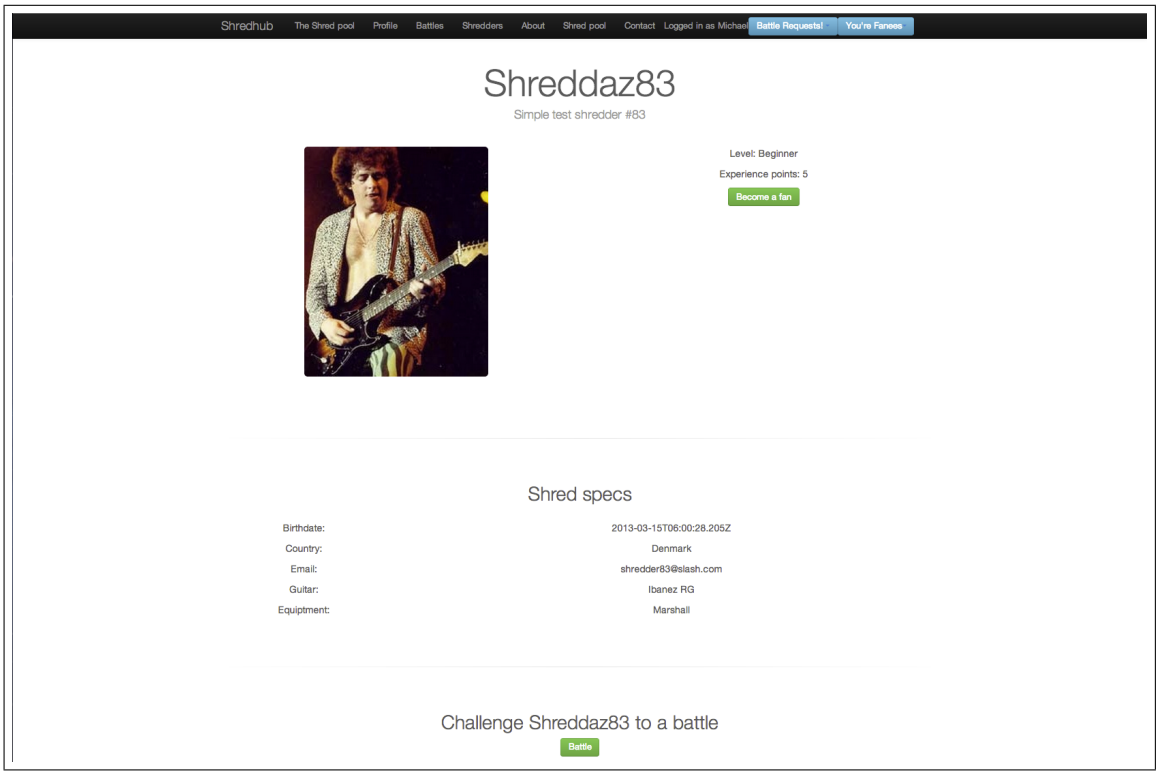

Figure 4.5: A shredder's profile page

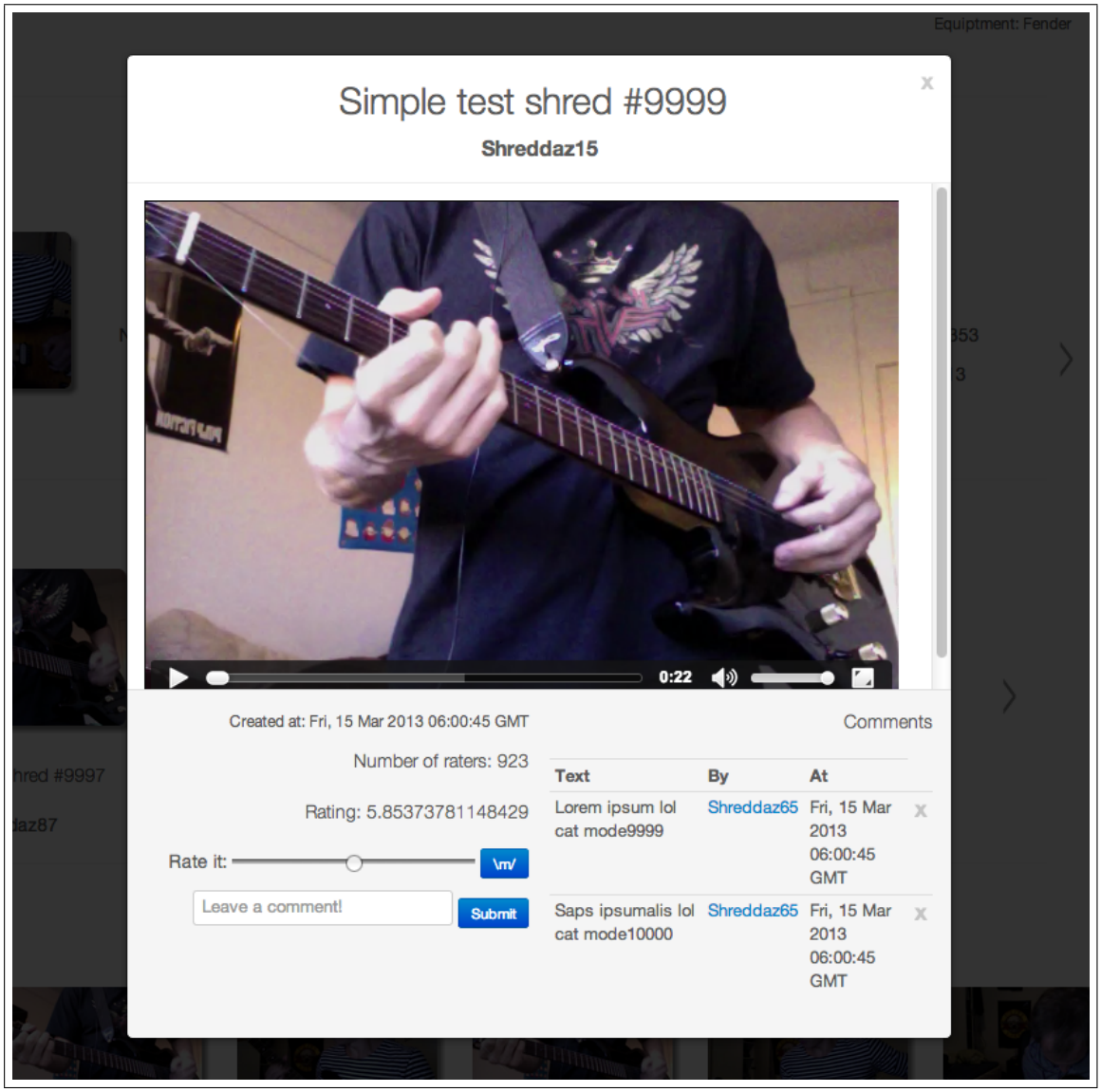

Figure 4.6: A pop-up window displaying a Shred

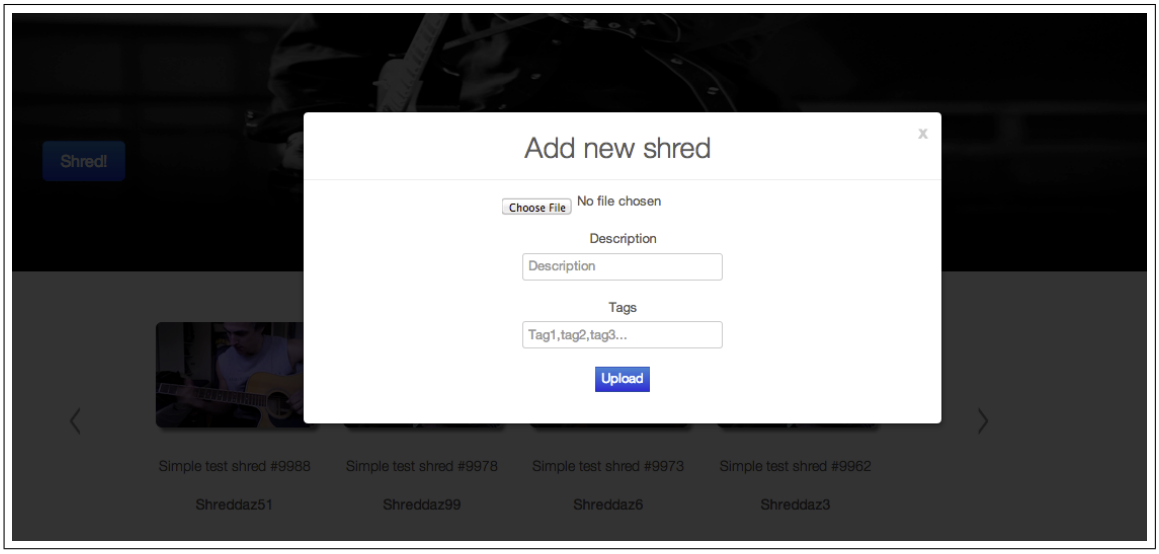

Figure 4.7: A pop-up that lets the User add a new Shred

Chapter 5

Architecture 1.0

5.1 Introduction

In this chapter we will look at the architectural details of Architecture 1.0, which is an implementation of Shredhub that conforms to *Reference-model 1.0*. The application is written in Java, and uses the SpringMVC framework. I could have chosen to use another technology like Ruby on Rails, Sinatra for Ruby or Microsoft's .net, considering these are all very popular Web application environments. However, because I happen to know the Java programming language very well, choosing a Java-based Web is preferable. Although there are other Java-based Web frameworks in addition to Spring, Spring was chosen because it is very easy to set up, it provides a wide collection of plugin extensions, it implements lower layer protocols like Http communication in a highly efficient manner, and most importantly, for the relevance of this thesis, it is one of the most popular Java-based Web frameworks.

The application runs on Apache Tomcat, which serves as both a Web server, and an application server. The database is implemented with PostgreSQL. It runs on a database server which for simplicity is deployed on the same physical machine as Tomcat. Here I could also have chosen a different database technology, for instance MySQL or Oracle SQL. However, I chose PostgreSQL because I have experience with the technology, it also has a lot of good and available documentation, and it is a very popular database choice for modern Web applications. [**popularDB**]

5.2 Architectural Overview

Architecture 1.0 is a backend-oriented Web app. All of the application's logic happens in a Web application that runs on an application server. In respect to *Reference-model 1.0*, the application is separated into three software layers with different responsibilities; a presentation layer, a domain logic layer, and a data source layer. I have chosen this separation of concerns in order to facilitate application maintainability and flexibility. I could have chosen to implement everything in two, or even one layer,

but this would resolve in classes having very many responsibilities, and possibly lots of code duplication. The layers cooperates by delegating responsibility. This means that the presentation layer implements client-server specific concerns, and delegates business logic specific operations to the domain logic layer. This layer implements business logic, and delegates to the datasource layer whenever it needs to address the database or some other data resource. The results from the operations done in the datasource layer bubbles up the layer stack all the way back to the presentation layer.

The Web app depends heavily on Http sessions to maintain user-state. When a user enters www.Shredhub.com, SpringMVC generates an object (called the HttpSession object) who's lifetime lasts throughout the user's session with the app. This object is used as a container for storing state information.

The application's front-end consists of a set of HTML pages that we will refer to as **Views**. The views are implemented with the JSP temple language technology, and are turned into HTML pages with the template view pattern. The client user primarily communicates with the app through three different interaction schemes:

1. Links (anchor tags)

2. HTML forms

3. Buttons or text input-fields that are picked up by JavaScript handlers

For all of the different user-interaction schemes in listing 1 and 2, there will be a corresponding function (called a controller handler) on the backend. These actions always result in a new view being rendered and returned to the client. Interaction scheme 3 only occurs a few times on the app, in special cases that requires highly responsive behavior, in which a server round-trip must be avoided. This is managed by Ajax calls that are implemented in the views.

In the following sections we will look into the implementation details of the source code. We will discuss the problems that occurred along the way, choices that were made, and potential alternative solutions. The discussion is divided into three; one part for each software layer in the application.

5.3 The presentation layer

The presentation layer is the first entry point in the application. Its responsibility is to handle client interactions, meaning it will handle authentication, session management, and input validation. The presentation layer is also responsible for knowing what operation to call in the domain logic layer, and it generates the views that are sent back to the client. It is built with the Model-View-Controller pattern, because this creates a nice and coherent separation of concerns.

5.3.1 Authentication

The first entry point in a SpringMVC application is a set of interception filters. In Architecture 1.0 these filters are configured to handle authentication, access control and a bit of input validation. Users must to be authenticated in order to use any of the pages on Shredhub except the login page. Most of the authentication and access control handling is set up to be handled automatically by Spring, through the Spring security framework [59]. Spring provides a lot of different authentication protocols, both based on standardized security protocols, and customized solutions made by third parties. For simplicity, I have chosen to use a Html form-based authentication mechanism that relies on a username, password and security-role. Other common authentication solutions used in Web apps are HTTP BASIC or HTTP Digest, or HTTP X.509 client certificate exchange. However, I find that form-based authentication fits the simple scope that has been chosen for authentication in this thesis, and it conforms to *Reference-model 1.0*, because it relies on server's session handler.

The form based authentication process works by letting users enter a username and password in an HTML form on the login page. On form submit, the request is picked up by Spring's interception filters, which will look in the database for a Shredder with the given username, password. The database row for a Shredder also contains a column that represents the Shredder's user-security-role. However, for simplicity there is only one role in this app, which is the one that gives access to everything. If a row with a matching username and password is found, the framework will grant access to the user, and the user will now have access to the whole Web app. To avoid having to re-authenticate for every subsequent request, Spring will behind the scenes maintain a security-context object that is connected to the User's HttpSession. The security-context object simply indicates that the user has successfully logged in once, and is allowed to perform the given request.

5.3.2 State Management

In respect to *Reference-model 1.0*, state is to be maintained on the server.

The presentation layer is responsible for managing state associated with a User when he navigates around the app. In Architecture 1.0, this is implemented by using Spring's HttpSession object. The Spring IOC container creates one HttpSession object for every current User, which naturally is scoped at session-level. Considering that Spring offers readily available in-memory objects scoped at session-level, makes it very appropriate to use such objects as caches for data that is frequently accessed. In the Web app, I put state data either in objects that are maintained by the IOC container and scoped at session-level, or I put them directly on the HttpSession object, using a method called *setAttribute(String key, Object value)*. The separation is a matter of separation of concerns; HttpSession object maintains meta data concerning the User (profile info, battle requests, battles, fanees etc), while other session-scoped objects

maintains data regarding the User's page activities (e.g current shred-row, current shredder-page etc).

Data that is used to populate views on the server has to be fetched from the database. Many of these database calls can be avoided if some of the data is stored in memory. The data could for instance be a particular set of Shreds that must be fetched especially quick in order to achieve responsive behavior, User data that is displayed often, e.g the User's name, or a list of battle requests which is meant to be visible on the top navigation-bar at all time. Shredhub needs to be very efficient and responsive, therefore solutions to avoiding repetitive and redundant database lookups has been prioritized in this architecture.The problem however, is that there is a tradeoff in how much data should be kept in memory, as maintaing too much memory might make the application slow, and in worst case lead to out of memory exceptions. Also, and this is a special case for typical Web 2.0 applications, data tend to change frequently, and users naturally want up-to-date views of the data. Hence, content regarding the Shreds and the newest Shredders on Shredhub, new fan-connections and other data that frequently changes, for simplicity shouldn't be cached. However, alternative solutions that makes it possible to cache such data is to some extend possible, for example by implementing push-based public-subscribe service that signals the cache to update whenever an update is made. Or alternatively a pull based solution where some service frequently pulls the database for new updates. A very simple proposal for such a solution is given later in this section.

Given below I have addressed two different collections of data objects that are frequently accessed in the application, together with an indication of how frequent the object changes, how often it is accessed, and the decision of wether they are cached in the User's session or not:

1. Data that is fetched in each Http request:

Data object	**Changes**	**Displayed**	**Cached in session**
The User's user-name	Never	Always	Yes
The User's fanees	Moderate	Always	Yes
List of battle-requests for the User	Moderate	Always	Yes
The User's unique ID	Never	Always	Yes

2. Data that is accessed frequently

Data object	Changes	Displayed	Cached in session
User's profile details, like address, guitars, age, email, birthdate, etc	Seldom	Moderate	Yes
The User's current battles	Moderate	Often	Yes
The rows of Shreds that are displayed in the shred-pool	Often	Often	Partly
Page navigation data, e.g page numbers and Shred-row numbers	Moderate	Often	Yes
The shred-news list	Often	Often	No
Shred, Battle or Shredder the User just looked at	Moderate	Often	No

In general, data objects that are frequently accessed and that is special to the User, should, and is being cached in the session.

Table 1 : The reason I store everything in this table is because this data is accessed very frequently, it only addresses the User, and it doesn't change often. All of these objects are fetched when the User logs in, and for simplicity, I never pull the database for updates concerning this data. Hence, for instance if the User gets a new battle request during the session, it won't be visible until the next time he logs in. This is simply because I didn't have time to implement a pull/push based service, and I consider this behavior good enough for this thesis.

Table 2 : In the second table I have chosen to store the User's profile details, the User's current battles, and navigation data. The latter is just a set of small integer numbers that controls for example the page number for the list of Shredders the User is currently at, or which row number in a particular Shred-row on the shred-pool is the User currently at. This could also be stored in the Urls, however I chose to keep the Url's clean of state information, and instead maintain this state info on the server, as the Url option to a higher extend addresses *reference-model 2.0*. The data objects mentioned here do not change very often, and are most likely small in size. These are also fetched when the User logs in, and aren't updated, unless update is made by the User himself. This could be for instance if the User updates his profile details, adds a new fanee or starts a new battle. In this case I eagerly write to the database, and at the same time update the cache.

The set of Shreds that are displayed in the shred-pool are only partly cached: the shred-pool is made up of multiple rows of Shreds. Each row consists of 3-5 Shreds, depending on which Shred-row it is, and the User

can click a next button change the particular row to a new set of Shreds. Now, for each row, the server fetches a set of 20 Shreds from the database, and maintains these in a session-cache (or buffer). Whenever the User clicks on the "next" button in a row, the server checks the cache for that row to see if the next row of Shreds lies within the buffer. If they do, the row is moved one row-size, and this new row of Shreds are displayed. If not, the server fetches another 20 Shreds from the database and displays the first row from the new buffer. By using this buffer of Shreds for each row, the server avoids many calls to the server, which is very important in order to get quick and responsive behavior when the User clicks the next button. The buffer could also be bigger, but then again there is a tradeoff in how big the cache can be without influencing performance.

The last two rows in table 2 concerns data that changes often and should always be up-to-date. In this case it is not worthwhile caching anything, considering the data must always be fetched eagerly from the database in order to be sure to have up-to-date data. The last row concern objects that could actually be saved in the cache, considering there is a chance the User might want to look at them again shortly, and in cases where the data is not likely to have been update yet. However, again, for simplicity I have chosen not to cache.

A Simple Cache Proposal

As caching is very important for maintaining good performance , a simple caching solution for Shreds and other data that frequently changes is given in the following outline: *The caching scenario works by having a Java container object scoped at session-level, that maintains a HashMap of cached objects. The objects are be referenced by their database ID, so that whenever a request comes in for either a Shred, Battle or a Shredder object, the client of the cache container checks if the object with the Id is there. If so, it is returned, if not, the object is fetched from the database, put in the HashMap and returned to the requestor. To maintain updated values, there could be a background process that is triggered every time the cache is checked. For every object in the HashMap, the background process checks the database for the object with the given Id, and compares the value of the time-of-last-update field. If the object in the database has a newer time value, the database object is swapped with the outdated object in the cache. Spring provides many different implementations of an interface called TaskExecutor, which is able to perform background jobs both asynchronously and in concurrent fashions. An alternative to fire of the background job every time the cache is accessed is to use a Scheduled job that executes the background process every n seconds.*

This cache could be used both for a User's session, such that data that concerns the user himself (for example the set of Shred, or Shredder recommendations that are displayed in the shred-pool), or it could be maintained as a general cache for every user. This would maintain data that is frequently accessed by everyone, for example the list of the top rated Shreds that is displayed in the login page and in the shred-pool. Unfortunately I have not had the time to implement such a solution.

It is worth mentioning that there are many alternatives for how data can be cached in memory in order to avoid consulting the database. Another approach I could have taken is to use a third party service like Memcached,[**memcached**] which is a highly efficient in-memory storage of data that can be placed on the same machine as the application server, or (as recommended) distributed to multiple machines. This would extend the caching capability of Shredhub. Unfortunately, I did not have the time to integrate such a service for this thesis.

5.3.3 Input Validation

Input validation is both part of the presentation layer, which addresses form-input rules, and the domain logic layer, which enforces business rules of the input data. In the presentation layer, validation is usually the first thing that happens once a Url request enters the server. Validation is always performed by applying positive filtering, meaning I specify what's allowed, and forbid everything else. Another approach is to do negative filtering where the input scanned for illegal patterns. However the latter approach is not as secure because it is hard to imagine all possible attack-forms.[62] Also, new forms of attacks might be invented in the future. However, positive filtering has the downside that it might be to restrictive.

Mainly I have two ways of verifying user input in the application. One of them is to enforce validation rules directly onto the variable in the domain object classes, I.e the Shredder class will contain validation rules concerning its data variables, the Shred class contains validation rules concerning its variables etc. This feature is something that's provided by Spring, using keywords in the domain classes that are automatically picked up by Spring. This validation is performed whenever the User creates a new domain object, such that the validation is executed before control is handed to a controller. The other way validation is done is by manually checking the Html form-input illegal input, or access rules. For example the controller often checks that the User stored in the session is allowed to perform a given business operation, or that a video file is sent when the User wants to add a new Shred.

An example of validation rules applied to a domain object is displayed in the code below. This shows the use of regular expressions and other rules to define the format of the data that is allowed for a Shredder. The annotations (referenced by the "@" character) are picked up by SpringMVC whenever a new Shredder object is created. Spring will then check that the data entered matches the rules, if not, an error is generated. Hence positive filtering.

```
1  public class Shredder implements Serializable{
2
3          @Size(min=3, max=20, message=
4              "Username must be between 3 and 20 characters long."
                   )
5       @Pattern(regexp="^[a-zA-Z0-9\\s]+$",
6        message="Username must be alphanumeric")
7       private String username;
```

```
8
9           @Pattern(regexp="[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z
                ]{2,4}",
10                  message="Invalid email address.")
11          private String email;
12
13          @Size(min=6, max=20, message="Passwords must be between 6
                and 20 characters long.")
14          @Pattern(regexp="^[a-zA-Z0-9\\s]+$",
15                message="Passwords must be alphanumeric")
16          private String password;
17
18          .... // other fields
19
20          }
```

Before I went with this approach, I tried to implement negative filtering. Here, I created a custom validator class that would perform negative filtering on the user data. The validator's validation function is executed before the appropriate controller handler is called. The validator function would inspect the input to find potential illegal data. If any potential violation characters or patterns were found (for instance the sequence " ' OR 1=1; SELECT ...; – ", which is a typical SQL-injection attack) I created an error message that is delivered to the controller function that is to execute next. This approach was very cumbersome, because it had to test for many different text patterns, and also, it is hard to know every possible set of illegal inputs.

5.3.4 Controllers

Controllers are first-class citizens in the presentation layer, who's responsible for processing URL requests. These components represents the Controller part of MVC. Controllers are Java classes that are mapped to specific a specific Url pattern. A simple approach is to have one controller class that is used for every Url supported by the Web app. However, this is not very maintainable, as the class would grow exceptionally large, and have many responsibilities. Other solutions are for instance to have one controller class for every View, or one for every domain object. I have chosen to implement something in between. I chose to implement one controller class for each main resource/domain in the application, in addition to one controller for the login, and shredpool view (simply called the login controller). In the table below we can see all the controllers in Architecture 1.0, together with some example controller handlers and their respective responsibilities. Although not displayed in the table, note that each controller handler is mapped to a unique Url.

Controller handler	Responsibility	View returned
HomeController		
loginPage()	Handles requests for the login page. Fetches the top-rated Shreds from the database and renders the login page	The login view
loginSuccess()	Called when authenticating the User succeeds. Populates the session cache with data fetched from the database	None: redirects to www.shredhub.co
theShredPool()	Fetches from the database the set of Shreds and all the shred-news that are to be displayed in the shred-pool.	The shred-pool view
showShredInShredPool()	Given a Shred id, the Shred is fetched from the database and renders the shred-pool view so that it displays the Shred as a pop-up window on top of it	The shred-pool view
ShredController		
createShred()	Creates a new Shred, saves to the database	The shred-pool
postComment()	Ajax supported function that given a shredId and comment-text, adds a new comment to a Shred where the comment-owner is set to the id of the User (stored in session)	None
ShredderController		
getShredders()	Fetches the next page of 20 Shredders that are to be displayed in the list of Shredders view. The page number is maintained in the session object	The shredders view
followShredder()	Adds a new fanee to the User's list of fanees. Updates the session-cached list of fanees for the user	The shredders view
newShredder()	Creates a new Shredder. Stores it in the database	The login view
BattleController		
getBattle()	Fetches a battle object given a Battle Id	The battle view
newBattle()	Called when a Shredder accepts a Battle request that is being kept on the session object. Creates a new Battle object. Stores it in the database	The shred-pool view
getBattles()	Called when a Shredder accepts a Battle request that is being kept on the session object. Creates a new Battle object. Stores it in the database	The shred-pool view

This controller granularity is very intuitive and coherent, and it makes it easy to extend the application with more actions. For example if I was to implement a new function for the application, say, the use case: "remove a fanee-connection", this would naturally be implemented as a new controller handler function inside the ShredderController class. I could however, have created even more fine-grained controllers, say a ShredPoolController, a FanController, or even a BattleRequestController, but I feel the amount of controllers fits the application quite well.

Control Flow

In this section I will describe what happens in the controller when a Url request comes in. An example controller, and a controller handler is given in the code below. Extra comments are added to explain important things.

```
1
2  // Handles all Url requests for www.shredhub.com/shredder*
3  @RequestMapping("/shredder")
4  @Controller
5  public class ShredderController {
6
7    // Entrance to the domain logic layer
8    @Autowired
9    private ShredderService shredderService;
10
11   // Handles a url requests for www.shredhub.com/shredder/<
         someFaneeId>/?action=follow
12   @RequestMapping(value = "/{faneeId}", method = RequestMethod.
         POST, params = "action=follow")
13   public String followShredder(@PathVariable int faneeId, Model
         model, HttpSession session) {
14     Shredder user = (Shredder) session.getAttribute("user");
15     List <Shredder> shreddersFanees = (List <Shredder>) session.
           getAttribute("fanees");
16     try {
17
18       // Delegate to the business operation
19       List <Shredder> updatedFaneesList = shredderService.
             createFaneeRelation(user(), faneeId, shreddersFanees);
20
21       // Update the session
22       session.setAttribute("fanees",updatedFaneesList);
23
24       // Return the view that displays a list of 20 Shredders
25       return this.getShreddersAndReturnShreddersView(model,
             session);
26
27     } catch (IllegalShredderArgumentException e) {
28       // Something wrong happened in the business operation.
             Return the error-page view
29       model.addAttribute("errorMsg", e.getMessage());
30       return "errorPage";
31     }
32   }}
```

This particular controller handler is triggered when the User clicks an anchor tag with the link given in the comment above. The shredderService reference is the gateway to the domain logic layer. SpringMVC's IOC container will inject a concrete implementation of ShredServices, because the pointer is annotated with an @Autowired field, which is picked up by Spring when the application is started. When the request comes in, the controller handler fetches the user from the session, because the business operation needs the User's Id and his corresponding list of fanees to check if the User is already a fan of the Shredder with id=<faneeid>. Note that if this Url request would be a form submit that contains user generated input data, the controller would first check the result of the input-validation process done by the interception-filters. If the validation process found en error, the controller handler return an error page immediately, instead of calling the logic layer. Anyhow, if a business operation is called, the controller would upon return catch any exception that might have occurred somewhere down the line. For example if the User is already a fan of the other Shredder. Note that I could have chosen to run this check inside the controller handler before I call the business operation, but this is strictly speaking a business rule that belongs in the business logic layer. If an error is found, the a Model object is populated with the data that is to be rendered together with the error view. The Model object represents the Model in MVC. This object will be used to contain the data that is injected into the views that are sent back to the user. Going further, if the business operation that was called succeeded, the controller calls a function that does the following:

1. Set p = the current page number stored in the HttpSession object
2. Ask the logic layer to fetch a list of 20 Shredders, starting from page num = p
3. Put the result list on the Model : model.addAttribute("shreds", resultList)
4. return the String "shredders"

The returning String is picked up by SpringMVC's View Resolver which I have configured to map my Java Strings to names of Jsp files. Hence the View Resolver will look for a view names "shredders.jsp", or "errorPage.jsp" in case the error view is returned from the handler. How the views are generated is explained in the next section. The reason I do this last part in a separate function is because it is called by other controller handlers as well. Also, the reason I chose to return the shredders page, is because it is from this page the User initially clicked the follow link, and therefore the controller just returns the same page as the user was in.

5.3.5 Views

Views represent the V of MVC. In Architecture 1.0, Views are implemented with the Java Server Pages (JSP) technology. JSP is fully supported by

SpringMVC's template engine, and is a popular choice for dynamic view generation. The reason I chose JSP is because it has good support for many tag libraries, which extends the language features that can be used in the Views, and it makes it easy to integrate the domain objects' data fields into the view. Also, it has a nice and intuitive syntax, that is more Java-like then other template technologies (for instance Velocity or Freemarker). However, one could argue that choosing Velocity or Freemarker is a better approach because its syntax is more appealing to non-Java developers.

It is considered best practice to avoid implementing business logic in the views. **??** on page ?? View logic, on the other hand (code that generates the user interface), should only be done in the presentation layer, because it has got nothing to do with either the domain logic layer, or the datasource layer in terms of code-responsibility. To blend these responsibilities together would result in tight couplings between how the page looks like, and the data it operates on. This is one of the reasons I chose to implement the MVC pattern, because it nicely separates these concerns, making it easy to change the view without harming the business logic, and likewise to let the models be unaware of its presentation, so the presentation and models can change independently. Note that the models are implemented in the domain logic layer, but are used by controllers and views in the presentation layer.

Main Components

There is one View for each page in Shredhub. These are:

1. The login view
2. The shred-pool view
3. The shredders view
4. The shredder view
5. The Battle view

Also, there are some views that are re-used in the above views:

1. the header view
2. the footer view
3. show shred view

Each of these views are implemented as a jsp file, e.g login.jsp. The last three views in the list are injected into the other views using special JSP syntax, in order to avoid view duplication. The views contain static HTML tags that never changes, some Ajax functions written in JavaScript, and external links to CSS files and JavaScript libraries. JSP tags are used to inject the domain objects into the View by referencing to the Model object that is populated with data in the controllers. A simple example of the shredders view is displayed below:

```
<html>
<body>
  <jsp:include page="header.jsp" />
  <h1>Shredders</h1>
  <form action=''searchForShredder''>
    <p>Search for a Shredder </p>
    <input type="text"/>
  </form>

  <c:forEach items="${shredders}" var="shredder" varStatus="i">
    <div class="shredderInList">
      <img src="<c:url value="/resources/images/profiles/"/>$
                  {shredder.profileImagePath}"  alt="Profile img
          ">

      <a href="<c:url value='/shredder/'/>${shredder.id}">${
          shredder.username}</a>
      <p>${shredder.description}</p>
      <p>Level: ${shredder.level}</p>
    </div>
  </c:forEach>
  <a href="<c:url value='/shredder/'/>nextPage">Next</a>
  <jsp:include page="footer.jsp" />
</body>
</html>
```

As we can see, the special template syntax is indicated with the "<c:" and "".*Thectagsrepresentsthingslikeloopsandconditionals,whilethe* tag directly refers to objects that the controller handler has put on the Model object (as we saw previously). Here, we use a c:forEach to loop through a list of shredders that were given to us in the controller that returned this particular view. After a controller handler finishes, and the View Rendered gets this view together with a Model object, it compiles the view by executing the special JSP syntax, and outputs a fresh HTML page. SpringMVC sends the Html page back to the client.

The next example shows the one and only place in the app where Ajax is used to implement dynamic in order to avoid fetching a complete view from the server. The example shows the a simplified version of how a Shred is displayed. Now, if the User rates or comments the Shred, the new result (I.e updated rating value or the new comment) has to be displayed very quickly in order to achieve proper responsive behavior. This could be solved the usual way by using form-submits, but in this case, this is not good enough, because it results in a complete page refresh in the browser. Everywhere else in the app however, regular form submits are ok.

```
  <srcipt type=''text/javascript''>
    function commentShred(shredId, commentText) {
      // Create the url that calls the controller handler on the
          server
      var baseUrl = "<c:url value='/shred/'/>" + shredId;
      var url = baseUrl + "/comment/?text=" + commentText;

      // Send the Ajax request to the server as a Http post
          request.
```

```
8         // The result from the server is the Shred with the list of
              comments
9         // updated with the new comment
10        $.post(url,
11
12        // This function is called when the server's response gets
              back
13        function(shred) {
14          // get the last comment from the shred, I.e the one the
                User just created
15          var lastComment = _.last(shred.shredComments);
16
17          // Create a comment as an html string,
18          // that is to be injected into the DOM tree
19          var htmlString = '<tr><td>' + lastComment.text +
20                      '</td><td>' + lastComment.commenter.username +
21          '</td><td>' + new Date(lastComment.timeCreated).
                toUTCString() + '</td>'+
22          '<td><button type="button" class="close"
23           onClick="deleteComment(' + last.id + ', ' + data.id + ');
                " >x</button></td></tr>'
24
25           // Append the Html to the table of comments
26          $('#commentTable tbody').append(htmlString);
27        });
28    }
29
30    function rateShred(shredId, commentText) {
31      // I have omitted the source code for this example,
32      // But it's very similar to the function above
33    }
34  </script>
35
36    <div class="videoView">
37      <video id="videoInModal" src=''<c:url value="/resources/videos
            /"/>''$                    {shred.videoPath''</video>
38        <p> Created at: ${shred.timeCreated} </p>
39        <p">Number of raters:${shred.rating.numberOfRaters}</p>
40        <p>Rating:${shred.rating.rating}</p>
41
42        <p>Rate it:
43         <input type="range" id="rateValue" min="0" max="10" name="
              rating" value="5">
44        <button id="rateButton"
45          onclick="rateShred($('#rateValue').val()); return false;">
46        Rate</button>
47
48        <p>Write a comment</p>
49        <input type="text">
50        <button id="commentButton"
51        onclick="commentShred($('#shredCommentText').val());
52         return false;">
53         Comment</button>
54
55        <h3>Comments</h3>
56        <table id="commentTable">
57        <thead>
58          <tr>
59            <th>Text</th>
```

```
60            <th>By</th>
61            <th>At</th>
62          </tr>
63        </thead>
64        <tbody>
65          <c:forEach items="${currShred.shredComments}" var="c">
66            <tr>
67            <td> ${c.text} </td>
68            <td> ${c.commenter.username} </td>
69            <td> ${c.timeCreated} </td>
70            </tr>
71          </c:forEach>
72        </tbody>
73        </table>
74    </div>
```

In order to display the video, I have chosen to use the HTML5 tag <video>. This tag makes the browser fetch the video from the Url defined in the "src" attribute of the video tag, display a video frame, and provide a play button that the User uses to start the video. Another alternative to show videos is by using streaming technology, where raw bytes are transferred in real time while the user watches the Shred. This way the whole video doesn't have to be fetched before it can start to play. However, the performance benefits with choosing either of these approaches is out of scope for this project, so I won't dig into it.

Its worth noticing the *onclick* attribute on the rate- and comment buttons. These are JavaScript handlers that are triggered when the user clicks either of the buttons. When the onClick events are fired, the browser will call a JavaScript function (one of the two we see in the code example) that uses Ajax to submit the request to the server. The result from the server is the updated Shred object which is used to populate manipulate the DOM tree to display the updated value. As stated in previous chapters, the JavaScript functions are independent, and self contained. An unfortunate property here is that the JSP view now contains Html, JSP tags and JavaScript. Three languages in the same file, which makes the code somewhat messy. I could put the JavaScript files in a separate source file, but the JavaScript would still have to use both Html and Jsp tags.

5.3.6 Summary of The Presentation Layer

The presentation layer is built with MVC. Form submits and link actions are picked up by a specific controller handler on the server. The handler performs validation, state management, and delegates to a business function in the domain logic layer.

Models are implemented in a lower layer, but are heavily used by the presentation layer. After a business operation is performed, the Controllers are responsible for choosing which Model objects to send to a View, and the Views can access the data in the Model through special JSP syntax.

The presentation layer depends heavily on state implemented as session-scoped objects, but due to the nature of Web 2.0 applications, which requires data to have up-to-date values at all time, there isn't very

much data to cache. Also, I acknowledged that there should have been a dedicated caching system in this architecture, but this was out of project scope to implement.

5.4 The Domain Logic Layer

The domain logic layer is the part of the application that receives a specific action from the controller, and performs the necessary business logic needed to complete the action. Now, since the responsibility of the logic layer is to implement the business logic in the application, it is important that this software layer is flexible. Flexibility, means that it can be relatively easy to add new features without harming anything else in the source code, and it should be easy to modify the already existing code. To achieve this, I needed a coherent design, preferably built with design patterns that gives a proper structure to the software architecture. To organize the layer, I have chosen to use the service layer design pattern [55]. This pattern creates a boundary into all of the application's business logic operations, by dividing the application into logical abstractions. Each abstraction represents the business operations that operates on a particular resource, or domain in Shredhub (e.g a Battle, a Shredder, or a Shred). However, this is not to be associated with the application's domain objects which concerns the domain resources' data, not functions. Instead, the service abstractions wraps the set of operations that are supported for each resource. Hence, each service class has a set of service functions. The list of service classes with some essential service functions are given below:

1. BattleService
    - getBattleWithId
    - getOngoingBattlesForShredderWithId
    - acceptBattleWithId
2. ShredderService
    - addShredder
    - getShredderWithId
3. ShredNewsService
    - getLatestShredNewsItems
4. ShredService
    - getFanShreds
    - getAllTags
    - getShredsForShredderWithId

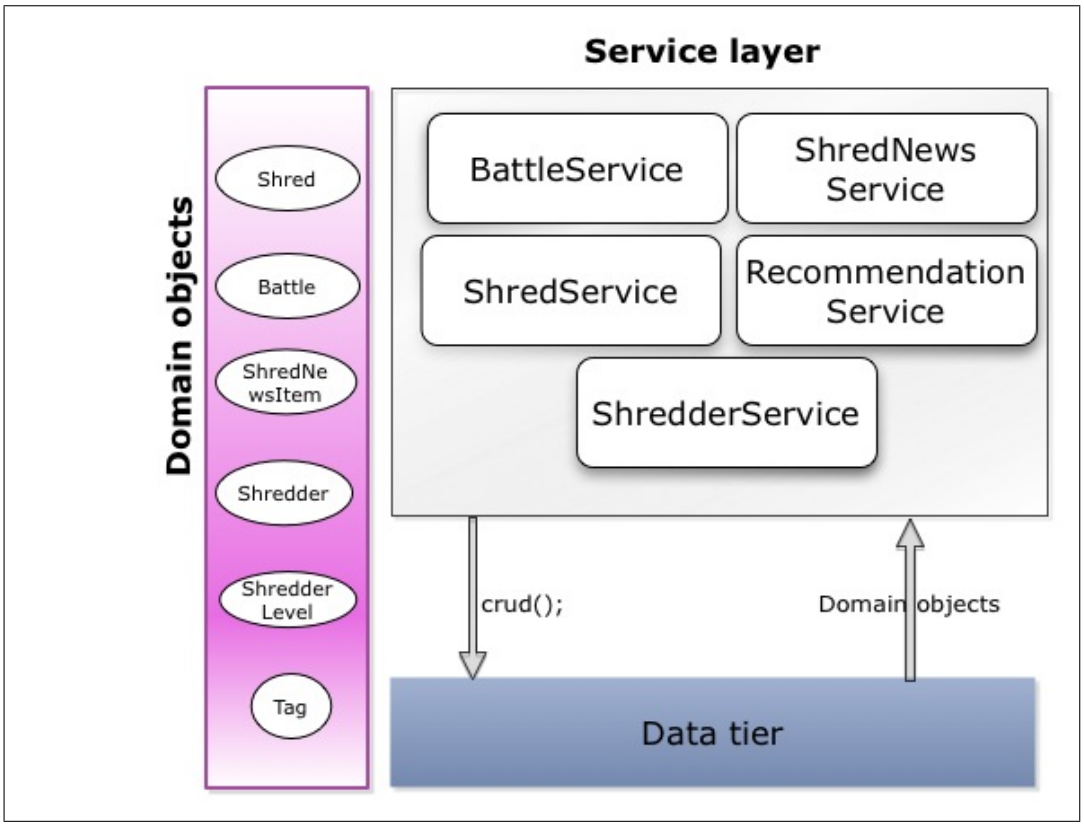

Figure 5.1: The domain logic layer and its connection to the datasource layer

5.4.1 Service Functions and Domain Objects

The service functions are called by the controllers in the presentation layer. An overview of the domain logic layer and its responsibility relative to the data source layer is given in figure 5.1 on the preceding page. The figure shows all the service abstractions, and the set of domain objects in Architecture 1.0. The domain objects represents the application's core resources, implemented as simple Java classes with no logical functionality, just attributes with accessor methods. The domain objects I have chosen to implement are Shred, Shredder, Battle and BattleShred. These objects have references to each other, and are used across all three layers in the application, making them the one means for communicating the domain in the application. The logic layer sends domain objects to the datasource layer in order to persist them. When the logic layer needs data from the datasource layer (e.g the database), the datasource layer will respond with domain objects.

An alternative architecture I could have chosen instead of the service layer pattern, is the table module pattern [55]. In the table module pattern, all the SQL rows in the database gets a separate class that is responsible for performing the operations that are made on each row. However, this approach is very tight coupled to the database. Another alternative is to use the domain model pattern, which is often implemented in Ruby on Rails apps. Here, the domain objects would be responsible for performing business logic on themselves, and manipulate the database. This is a good approach, but the domain objects might end up being very big, with lots of responsibility. I prefer the solution of having a separate service class for each logical application abstraction, where the service functions performs all the necessary business logic for a given action, and the domain objects are plain data holders. However, a hybrid is also possible, where the domain objects implement parts of their logical behavior. However, I prefer to separate these concerns.

A final thing to point out is that because some service operations perform multiple subsequent database transactions, each such service operation must be atomic. Spring achieves this behavior very elegantly by letting classes be declared as transactional, meaning every service operation in that class will be transactional (i.e have atomic behavior). In effect, if a database operation fails during a set of multiple database operations, the whole transaction is rolled back to the initial state. Hence atomicity (the A in ACID) is provided.

5.4.2 Data Flow

Most operations in the domain layer follows the same structure; some data is fetched from the datasource layer, this data is manipulated together with the data it got from the calling controller handler, and the result is written back to the database. Also, sometimes the newly updated data is returned back to the controller, so that this can be rendered in a new view. If an error occurred along the way (for instance if illegal data was sent from the

controller), an exception is thrown and picked up by the controller so it can return an error view. An example of a typical business operation is given in the code below:

```
1   @Service
2   @Transactional( readOnly=true )
3   public class ShredServiceImpl implements ShredService {
4
5     // Data source reference
6     @Autowired
7     private ShredDAO shredDAO;
8
9     /**
10    * This adds a new rating to a Shred.
11    * When a shred is rated, the shred will gain a higher total
          rating, and
12    * the shredder who made the shred also achieves
13    * more experience points. Note that I don't check if the one
          who rates the
14    * Shred is the one who created it. This is a business rule
          that should be implemented
15    * here, inside the business operation. However, for simplicity
           I have avoided it.
16    */
17    @Transactional ( readOnly = false )
18    public void rateShred(int shredId, int newRate) throws
          IllegalShredArgumentException {
19      Shred shred = shredDAO.getShredById(shredId);
20      if ( shred == null ) {
21        throw new IllegalShredArgumentException("Shred with id: "
              + shredId + " does not exist");
22      }
23      if ( newRate < 0 || newRate > 10 ) {
24        throw new IllegalShredArgumentException("Illegal rate
              value!");
25      }
26
27      // Here I could use the domain model pattern so that the
28      // shred object itself knows how to set its own rating.
29      // But I choose to follow the service layer pattern, where
            all the logic
30      // is implemented in this service operation
31      ShredRating currentRating = shred.getRating();
32      currentRating.setNumberOfRaters(currentRating.
            getNumberOfRaters() + 1);
33      currentRating.setCurrentRating(currentRating.
            getCurrentRating()+newRate);
34
35      // store the result in the database
36      shredDAO.persistRate(shredId, currentRating);
37
38      // Fetch the complete owner (Shredder) object from the
            database
39      Shredder shredder = shredderDAO.getShredderById(shred.
            getOwner().getId());
40
41      // Uses a utility class that is shared by all the service
            classes
42      // in order to update the shredder level
```

```
43          UpdateShredderLevel usl = new UpdateShredderLevel(shredder,
                newRate);
44          usl.advanceXp();
45
46          shredderDAO.persistShredder( shredder);
47      }
48  }
```

This function is called by the controller handler that receives requests for adding a new shred rating. At first, it tries to fetch the Shred from the database, before it performs some input validation on the data it received. If all went well, the service function will set the new rating and persist the result back to the database. Then it will fetch the Shredder who initially created the Shred in order to increase the Shredder's experience points. Finally it will persist the Shredder back to the database. Note that just like the presentation layer did with the domain layer, the domain layer delegates every persistency operation to the data source layer. The data source layer is reached through special references called DAOs. This is a nice separation of concerns, because the service functions only have to worry about the business rules, not how the data is persisted. Also, note that the operation in this example has to be atomic, because the function has multiple subsequent calls to the database. For example, if an error is made during the call to *shredderDAO.persistShredder(shredder)*, the whole transaction will be rolled back to the state that was before *rateShred* was called.

5.4.3 Summary of The Domain Logic Layer

The domain logic layer implements the domain of the application. The layer is divided into two; a service layer that implement business logic operations, and the domain objects which wraps the domain into self-contained data holders. The domain is divided into three abstractions; a Shred, Shredder, and a Battle class. The services are separated into a ShredService, ShredderService and BattleService class. The services forms a facade that is used by the controllers in the presentation layer. The services delegates to the data source layer for persistence.

5.5 The Data Source Layer

The datasource layer is the part of the application that receives a particular CRUD command from the domain logic layer, executes a SQL operation on the database, maps the result to a domain object and returns the result back to the business logic layer. I have chosen to use a PostgreSQL database, because it is open source, highly efficient, popular in the web industry, well documented, and most importantly, I know the database quite well. Also, I have chosen not to use an ORM mapping tool, but rather build Java functions that talks directly to the database using Strings as queries, and mapping query results manually to Java objects. There are many good ORM technologies I could have chosen to use, for example Hibernate,

and JPA, which would hide the complexity of serializing Java objects to SQL, and the opposite, and not having to deal with SQL. However, these technologies does not give me the control I need to fine-tune, debug and create flexible and complex queries. A tradeoff though, is that writing SQL with java statements tend to get messy, especially when the queries gets many and complicated. I do however value the control one gets by explicitly writing every query with SQL. Obviously performance is important in this project, which makes it a big benefit to be able to take advantage of PostgreSQL's proprietary features.

5.5.1 SQL Implementation

The Java driver that connects to, and sends queries to the database is called JDBC. Spring provides a nice wrapper around the JDBC framework through a class named JDBCTemplate. This class takes care of all the boilerplate code like resource management and exception handling that is required when operating with JDBC. The SQL tables that represents the three central domain objects in the application is showed in the example below:

```
1  CREATE TABLE Shredder (
2    Id          serial PRIMARY KEY,
3    Username    varchar(40) NOT NULL UNIQUE,
4    BirthDate   date NOT NULL CHECK (BirthDate > '1900-01-01'),
5    Email       varchar(50) NOT NULL UNIQUE,
6    Password    varchar(10) NOT NULL
7    Description text,
8    Address     text,
9    TimeCreated timestamp DEFAULT CURRENT_TIMESTAMP,
10   ProfileImage    text,
11   ExperiencePoints  int DEFAULT (0),
12   ShredderLevel   int DEFAULT (1),
13   Guitars     text[],
14   Equiptment  text[]
15
16 );
17 CREATE TABLE Shred (
18   Id          serial PRIMARY KEY,
19   Description text,
20   Owner       serial REFERENCES Shredder(Id),
21   TimeCreated timestamp DEFAULT CURRENT_TIMESTAMP,
22   VideoPath   varchar(100) NOT NULL,
23   ShredType   varchar(30) DEFAULT 'normal' CHECK (ShredType ='
         normal' or ShredType = 'battle')
24 );
25 CREATE TABLE Battle (
26   Id          serial PRIMARY KEY,
27   Shredder1   serial REFERENCES Shredder(Id),
28   Shredder2   serial REFERENCES Shredder(Id),
29   TimeCreated timestamp DEFAULT CURRENT_TIMESTAMP,
30   BattleCategory    serial REFERENCES BattleCategory,
31   Round       int DEFAULT 1,
32   Status      varchar(30) DEFAULT 'awaiting' CHECK (Status ='
         accepted' or Status ='declined' or Status='awaiting');
33 );
```

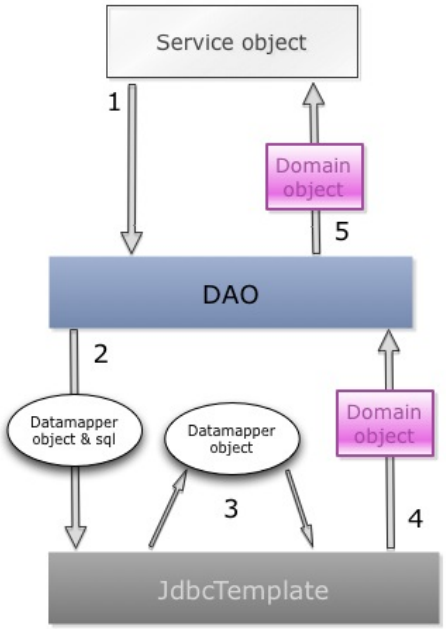

Figure 5.2: Data access pattern in the data tier

In addition there are many-to-many relations between Shredders and Shreds, Shredders and Battles, and Battles and Shreds (actually BattleShreds, but they almost identical). Its important to mention them, because it requires the data source layer to perform complex join operations when fetching data from the database.

Now, there are lots of other smaller tables in addition to these, but these are the most essential. To perform the CRUD operations, I have chosen to structure my data tier around the Data Access Object (DAO) pattern. In this pattern, separate DAO objects are responsible for performing the relational data mapping on behalf of a particular domain object. This way the domain objects has no clue on how to persist themselves. An alternative to this is to use the Active record design pattern, where each domain object contains persistence code. However I prefer to keep this behavior separated from the domain objects as they would grow quite large and complex if they where to contain all the necessary object relational mapping code. Figure 5.2 shows how object relational mapping is done in the data tier. Here's what happens when the domain logic layer asks the datasource layer to perform a CRUD operation (e.g a Read operation).

1. The logic layer calls a persistence function on a particular DAO object, for instance *shredDAO.getShredById(int shredId);*

2. The DAO class uses a JDBCTemplate instance (injected by the Spring IOC container) to delegate the boilerplate behavior, for example getting a database connection. The JDBCTemplate is also responsible for executing the database query itself, provided that it gets a SQL

statement from its caller. For instance:

```
1  @Service
2  public class ShredDAOImpl implements ShredDAO {
3
4    @Autowired
5    private JdbcTemplate jdbcTemplate;
6
7    public Shred getShredById(int shredId) {
8      String sql = "SELECT * FROM Shred s,Shredder sr WHERE
           s.Owner = sr.Id AND s.Id = ?";
9      return jdbcTemplate.queryForObject(sql, new Object[]{
           shredId}, new ShredMapper());
10   }
11 }
```

3. The JDBCTemplate is built with the template method design pattern, meaning it does callbacks to a mapper object provided by the caller. In the above code, the callback calls a function in the ShredMapper class. This class knows how to build a Shred object given the result from a database query that the JDBCTemplate executes. An example of how a ShredMapper class might look like is given below: Example:

```
1  public class ShredMapper implements RowMapper <Shred>{
2
3    public Shred mapRow(ResultSet rs, int rowNum) throws
       SQLException {
4      Shred shred = this.setConcreteShredder();
5      shred.setId(rs.getInt("id"));
6      shred.setDescription(rs.getString("Description"));
7      shred.setOwner( new ShredderMapper().mapRow(rs, rowNum) )
         ;
8      // Another template method!
9      this.addConcreteBehavior(rs, shred, rowNum);
10     return shred;
11   }
12 }
```

   Notice that I also use the template method pattern to enable customized mapper functionality that can be overridden by subclasses of the ShredMapper class. This is done with the BattleShredMapper, which extends the ShredMapper class, and hence overrides setConcreteShredder() and addConcreteBehavior().

4. The domain object created by the mapper is returned from the JDBCTempate back to the DAO object, which depending on the type of query might catch an exception to provide nice feedback to the service function. An example might be thrown by the DAO if a Shred with the given Id does not exist.

5. Finally the DAO function returns the domain object back to the middle tier.

5.5.2 Storing Video Files

Media content is a central part of the application, since the primary purpose in the application is to share videos of people playing guitar. The videos have a maximum size of (x) bytes, because Shreds are supposed to be short (less then one minute). If a user tries to upload a file that is bigger then this, then an error message is given to the user. As mentioned in the presentation layer section, the whole video is downloaded to the clients web browser before it can be played. It is therefore important to offer efficient uploading speed from the application's backend. As for now, I store the videos locally on the machine that is hosting the application, inside a folder that is publicly available for the users. This has some unfortunate drawbacks. One is that the users can directly access which ever files they want on the server, if they know the url for the video. The path to where the video's are stored can be seen if one inspects the HTML source code that is sent to the client's browser. The user could simply guess the name for a movie, and try to fetch it from the server. A second problem by having no protection of the videos is that it would be easy to do overflow attacks, where an attacker simply asks for a whole lot of videos until the server breaks down. A third problem is that storing the files on the machine that hosts the server requires a lot of storage space on that particular deployment server. This is undesirable, because one would not want to waste deployment space on something that could easily be stored elsewhere. Therefore I have explored various ways to store the media content, that is not being on the deployment space. One good example is storing the videos and images on Amazon ec3 cloud storage. This is service is free of charge up to some x amount of requests and y amounts of data. There are also other alternatives to amazon, like SimpleCDN and windows azure, and they all look very similar in both pricing and performance. Unfortunately, I have decided that this is simply too time consuming to set up for this thesis, and therefore I have had to content myself with the simple solution described in the beginning of this section.

5.5.3 How Much Data to Fetch

Another issue is the challenge with deciding how much data to fetch from the database when an object is requested. For instance when a request is made for a Shred, should the DAO function fetch the whole Shred object, fetch the Shred's owner, all the tag objects for the shred, all the comments and rating etc. This is a tradeoff decision, considering fetching everything requires many SQL joins and much data-mapper processing in Java. But it avoids having to fetch the server for more data at a later point in time, if more of the domain object has to be fetched. One approach is to eagerly fetch every table column and to populate every foreign reference, which would require a large amount of processing, very much data stored in memory, and much data returned back to the client that probably will never be used. The decision I have made is to implement CRUD operations that are customized for the controller handlers in the presentation layer. For

example, if a list of shredders is needed that are to be displayed in the shredders view, the read operation called in the database will populate a list of Shredders without their related list of fanees, shreds and battles. This is because these lists aren't needed in the Shredders view. On the other hand, if the shredder view is to be rendered, the database will populate the Shredder with all its fanees and all its shreds.

5.6 Summary of The Data Source Layer

The data source layer is responsible for manipulating the database. The database is built with PostgreSQL, where object relational mapping is manually built in Java using, instead of using an ORM tool. This requires a lot of Java code, but, being built with design patterns like the DAO-, and template method pattern, the code is very flexible and facilitates optimized object mapping and querying.

5.7 Summary

Architecture 1.0 is a Java web app built with SpringMVC. All the application's logic happens on the server, where the code is divided into three separate layers; the presentation-, domain logic- and datasource layer. The presentation layer is built with the MVC pattern, in which controller handlers handles Http requests, calls a business operation in the logic layer, and upon return, populates a Model object with data that is needed to create a View that is rendered to an Html page and sent back to the user. The domain logic layer implements business operations, and delegates the persistency handling code to the datasource layer. The application relies heavily on the Session object to maintain state.

Chapter 6

Architecture 2.0

6.1 Introduction

In this section we will look at the details of Architecture 2.0, which is an implementation of Shredhub that conforms to *Reference-model 2.0*. The application is completely implemented with JavaScript, both on the back-end and the front-end. The back-end is built on a Web server that is implemented using Node Js. Note that I could have chosen to use any Web framework for this as well and still have conformed to the principles in *Reference-model 2.0*. For example Ruby on Rails, or SpringMVC. However, the reason I chose Node Js is because The programming language is JavaScript, which gives me a pure JavaScript codebase, and in effect creates a more complete and coherent codebase. This also minifies the distinction between the front-end and the back-end. Even more, the framework does not automatically create Http Sessions, which is an important architectural principle in *Reference-model 2.0* where the back-end must be completely state-less.

The persistency layer is made with two popular noSQL solutions. One is a highly efficient Key-value store that serves to authenticate Users. It is built with Redis. The other is a MongoDB database which persists the rest.

The front-end uses various frameworks that helps improving the JavaScript codebase. These are Backbone for code-structuring, RequireJs for AMD, and JQuery for DOM manipulation.

6.2 Architectural Overview

Architecture 2.0 is a large-scale JavaScript Web application where most of the data processing and manipulation happens in the client's browser. The application is basically a complete JavaScript application that is sent to the client in the initial visit to www.Shredhub.com. Unlike Architecture 1.0, the application's state is maintained on the client, such that the back-end is completely unaware of any reoccurring User (session). Also, instead of the using a traditional SQL database, the back-end uses noSQL technologies for data persistence. For simplicity in this chapter, we will use the term

App to refer to the JavaScript application that runs in the client's browser, and we will use the term *API* to refer to the code that runs on the back-end.

In order to achieve a flexible structuring of the app's codebase, it is built with the MV* pattern. This pattern suits the application because it separates Models from Views, and it lets Views help out with controller specific logic. I could have designed the app around a traditional MVC architecture, but I felt this approach didn't quite fit with the application's needs. The reason is that MVC is very controller-oriented and structures the application around a set of controller handlers. However, in this app, Views aren't just Html files that contains some special syntax to render the Model's data. They are JavaScript objects that listen to events that occurs in their belonging part of a page. Therefore, considering that Views are both event-aware and responsible for creating dynamic Html pages, I wanted to let the Views be the first-order citizens and have them both be responsible for communicating with the Models, and control the data flow. Also, other architectural patterns would also fit the application quite well, examples being the MVVM pattern and MVP. These however, are even more view-logic oriented, something that might fit applications that contain a lot of updates of the visible data. Something Shredhub does not have in a high degree.

The App is bootstrapped from a single JavaScript reference in the initial Html, which simply asks the browser to fetch a particular JavaScript file that has a reference to a main function. This function loads in the rest of the necessary part of App, and from there on, everything is controlled by the App.

The app is composed of a set of loosely coupled modules, where each module contains a set of zero to many **Models**, **Collections** and **Views**. These are core entities in the application that together provides domain data, business operations, controller handling and view logic. Each module is a separate JavaScrip source file. The app uses the Asynchronous Module Definition pattern to lazily fetch scripts and define dependencies between modules. The alternative was to eagerly fetch the whole app in the initial request. However, considering this would require a large amount of JavaScript and Html files to be fetched over Http, the AMD solution was chosen as to avoid a potentially time consuming and resource demanding bootstrapping process. This way, unnecessary fetches can be avoided. Say for instance the User only accesses the Login page and Shredpool. In this case, the app will never load the Scripts and Html templates required for displaying the Shredder-, Battle-, or List of Shredders page.

In addition to Models, Collections and Views, there are three other central components in the app: The **Router** is a module that is responsible for managing data flow between the main pages in Shredhub, where the main pages being the ones that can be accessed through a unique URL. The **Session** is a module that offers a facade to manage session data. And lastly the **Mediator** which is a module that coordinates communication between the independent Views and Models. This facilitates a loosely coupled communication scheme inside the app, so that the communicating entities don't need to have many and complex relations to each other.

The Models are responsible for persisting and manipulating data. This is done by communicating with the API, using Ajax. The transport format used is JSON, which is a natural choice because no marshaling needs to be done the app, considering JSON is already a JavaScript supported data format.

The API is organized as a Rest API, meaning it publicly offers a set of services that conforms to the Http messaging scheme. In the API, the first-order citizens are the application's core resources (domains). In Architecture 2.0, these are Shreds, Shredders, Battles and BattleRequests. Hence, the API offers a set of self-contained operations that manipulates these resources. In respect to *Reference-model 2.0*, the API is stateless. To achieve this, every Rest operation has to contain all the information that is needed in order to execute that operation. In other words, no API operation depends on the result, or state from another API operation.

The API is responsible for communicating with the database. Now, because both databases are manipulated with JavaScript, no serializing of data is needed, which in effect simplifies the programming model. Figure **??** on page ?? shows an overview of the main software components in Architecture 2.0.

In the rest of this chapter we will go into details of the implementation of Architecture 2.0. This chapter will be somewhat different the the previous chapter which discussed Architecture 1.0. This is because Architecture 1.0 is very much back-end oriented, and therefore focused mainly on the back-end implementation. Architecture 2.0 is rather front-end oriented, but the back-end implementation is also very unique and relevant. Therefore, the following text is divided into a front-end section and a back-end section.

6.3 The Front-end

Also referred to as the *App*, the front-end is composed of a large JavaScript codebase and a set of Html template files that are used to dynamically generate Html. The codebase is divided into 6 independent components: Models, Collections, Views, Router, Mediator and Session.

6.3.1 The Bootstrapping Process

In Architecture 2.0, a fairly large JavaScript application has to be downloaded and initialized in the client's browser during the client's initial request to the Shredhub. I call this the bootstrapping process, because the client will ask for a small Html page that contains one single line of JavaScript. This statement is responsible for starting a recursive process that loads in the rest of the App from the server. In detail, the bootstrap process works like this:

1. The client visits www.shredhub.com and the server responds with a file called index.html

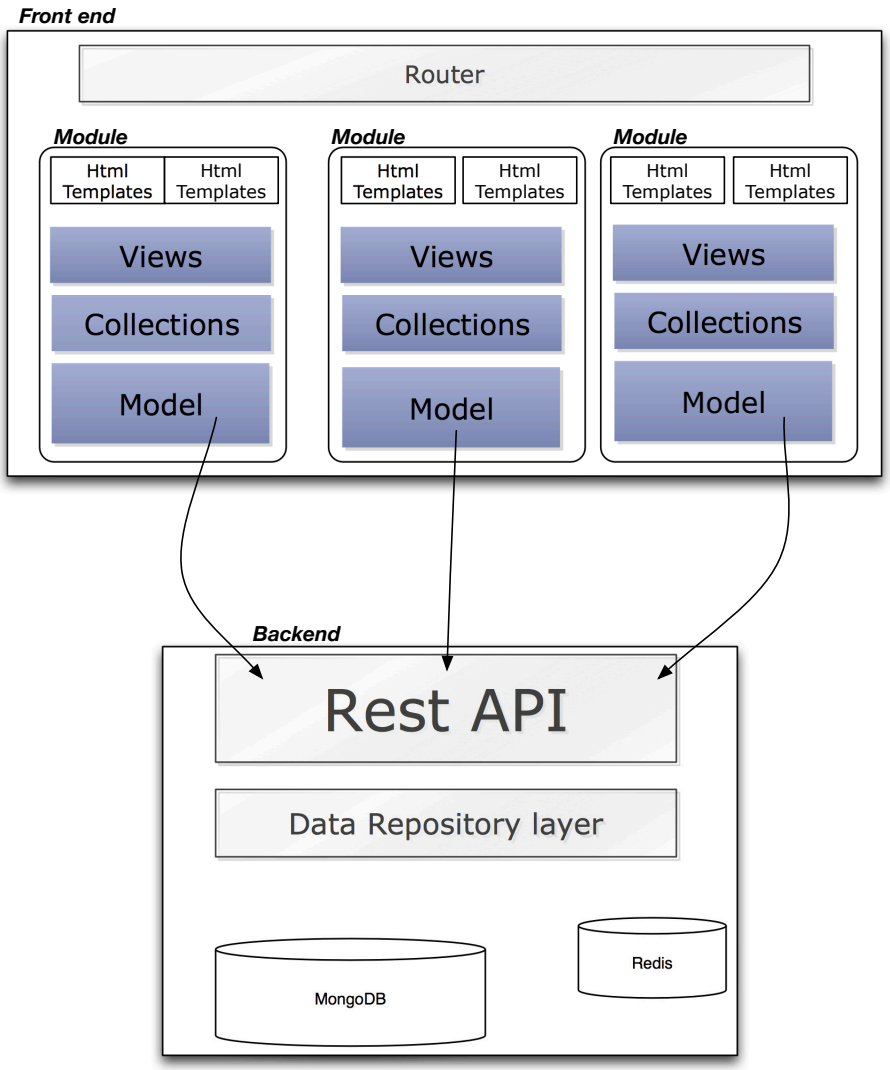

Figure 6.1: The main software components of Architecture 2.0.

2. index.html contains the line
<*script data-main="/app/config" src="/vendor/js/libs/require.js"></script*>,
which will fetch a JavaScript file called require.js from the server

3. require.js is a framework that implements the AMD specification. It will fetch a file called /app/config.js (outlined in the script statement above), which contains a reference to the **main** function.

4. The JavaScript file that has the main function is fetched from the server, and the main function is called

5. The main function is responsible for instantiating objects that will be globally accessible (that is, accessible through the whole codebase). This includes the Session, Mediator and the Router object. Also, a globally accessible object called *app* is created. This object will cache

Html templates in the browsers JavaScript heap memory, so that Html templates won't have to be fetched more then once.

6. When the Router object is initialized it will start listening to Url changes.

7. At the end of the bootstrapping process, the router will handle a request for the home page. This will result in the Home page View being created and rendered in the browser.

6.3.2 Router

The Router is the component that organizes routing between the Shredhub's main pages. Normally, every hyperlink request or form submit would make the browser send the request directly to the server. This however, is unwanted in Architecture 2.0, because the front-end is supposed to decide when and how to contact the server. This is where the Router comes in. The router is configured to listen to every hyperlink-event or form submit that is triggered in the app, so that when such an event is triggered, the router is notified, and it will call the **event.preventDefault()** function on the browser, which in effect tells the browser not to issue the Url request to the server. This way the Router has hijacked the request, and is now able to decide what will happen.

To some extend, the router works as a Controller from Reference-model 1.0, in that it receives a particular page request (for example www.shredhub.com/shredders), however unlike Reference-model 1.0, there is no controller handler for every possible request. There is just one handler for every possible hyper-link. Now, it so happens that I have chosen to have only 5 different hyperlink possibilities in my app, namely the ones that refers to any of the 5 pages offered in Shredhub. Hence there will be 5 controller handlers, or routes, as they are called in Architecture 2.0. After the router has hijacked a Url request from the browser, it will call the route handler for that particular Url. The handler's responsibility is very simple. It will remove the current View, and create the new View that will be displayed in the browser. The Router is implemented in a JavaScript file called router.js. It primarily contains a key-value mapping between a Urls a JavaScript function (the route handler). The url-handler mapping in router.js is showed below:

1. '*actions': 'renderHomeView' *//www.shredhub.com*

2. 'shredPool': 'renderShredPoolView', *//www.shredhub.com/shredpool*

3. 'shredder/:Id': 'renderShredderView', *//www.shredhub.com/shedder/<shredderId>*

4. 'shredders': 'renderShreddersView', *//www.shredhub.com/shredders*

5. 'battles/:Id': 'renderBattleView', *//www.shredhub.com/battles/<battleId>*

6. 'battles': 'renderBattlesView', *//www.shredhub.com/battles*

6.3.3 Models

The Models represents the domain resources of the application, which implements both business logic and data attributes. Hence they implement the Domain Model design pattern. I chose this as opposed to having a separate service layer. An additional service layer does result in more decoupling and separation of concerns (business operations and data holders in this case), but it also leads to more code and additional source code files. In this architecture, less code and files are to some extend preferable, considering these are data that must be transmitted over Http.

Models are built with the Active Record design pattern, meaning they are responsible for knowing how to perform CRUD operations on themselves. Hence, I avoid having to have additional modules that only concerns data source handling. Now, since CRUD'ing in this case means talking to the database, and the database lives on another physical machine that the the client; CRUD'ing happens via HTTP. Hence, Models will use Http to talk the the API on the server.

Like in Architecture 1.0, the models are totally unaware of how they are visualized in the Views. I prefer this solution here as well, because it separates two very different concerns, and the Models and Views can change independently of each other. I could have chosen to let the Models have closer connections to the DOM tree, so that whenever the Models change, they can notify the View(s) of the change, in effect making the Models View-aware. This design fits more in applications where Models changes frequently. However, in this application I have avoided letting the models change frequently on purpose, because it would require me to get frequent new information from the database, a problem discussed in the previous chapter. If I were to choose to let the Models change often however (e.g let them get automatic updates from the API when a new comment is made for a Shred), letting the Models be completely View-aware and automatically update the View on change events, would have been a reasonable solution.

6.3.4 Collections

Considering that the application has many "collections" of Models, e.g a list of Shredders on the Shedders page, multiple rows of Shreds in the Shred-pool page etc, it makes sense to encapsulate these Models in separate modules (Collections). This way, a Collection is a container for multiple coherent Models. The motivation for this, is that the Collections can also work as Active Records, in that they can be responsible for fetching a particular set of Shreds or Shredders from the database, regarding the collection of Models they control. For example a Shred-Collection representing a row of top-rated Shreds, would know how to fetch the top-rated Shreds from the API. Like with Models, I have decided to let the Collections not be View-aware.

6.3.5 Views

The set of pages in Shredhub are in Architecture 2.0 separated into logical coherent Views. These Views are JavaScript objects that holds a reference to a piece of Html which it is responsible for maintaining. A View contains zero or more Model and Collection objects, that represents the data that is displayed in the View, and are being used to delegate business logic operations to. The view's main job is to render an Html template together with its containing set of Collections and/or Models. In addition, the View is responsible for maintaining state for the particular Html portion of a page it represents, and to handle all user-events that happens inside that Html portion. A View can contain one or more sub-views, such that Views can form a tree of Views. Views are created ether by the Router when a page is to be rendered, or by a parent View, when it needs to create a child View that will renderer a smaller part of Html inside the current View.

There is one root View called the ScaffoldingView, which is the root view that contains visual elements that are always visible in the app (top navigation bar and footer). This View wraps one single child View. This child view (called the MainContentView), is the always the View that represents the current Shredhub page. For example, when the Login page is showed, MainContentView is set to point to the LoginView, and when the Shredpool is showed, MainContentView is set to point to the ShredpoolView etc. The set of Views in Architecture 2.0 is given in the table below:

A View's Data Flow

Views are created and removed using the Template Method design pattern. It is so that Views share a lot of common initialization code, and termination code. Therefore, I have chosen to create a super type *BaseView*, that all other Views extend. This way, when a new View is created and is to render its containing Html template(s), it executes a template method defined in BaseView. The creation of a new View is described below:

```
1  // The caller (Router or parent View) does:
2    1. call showView() with reference to view v and an html tag t
3
4  // In the showView function
5    2. if there is a current view that is to be overridden, remove
         this first
6    3. call v.render function
7
8    8. Upon return, inject the returned Html to the DOM, inside the
         t tag
9    9. Call view.postRender
10
11 // In BaseView.render:
12   4. fetch this.Html-template from server, or cache
13   5. call this.serialize() to populate a data object for the
         template
14   6. render the Html template together with the populated data
15   7. return the rendered Html
```

View Name	Responsibility
Scaffolding View	Contains the header and footer that is contained in every page. Always wraps one subview
Home View	Represents the login page. Wraps a set of subviews; a list of ShredThumbnail Views, and a ShredModal View
Shredpool View	Represents the Shredpool page. Wraps a set of ShredRow Views and a ShredModal View
Shredders View	Represents the list of Shredders page
Shredder View	Represents the Shredder page
ShredRow View	Represents a particular row of Shred thumbnails. Maintains state for the row, so that it knows how to advance to a new row the same Collection of Shreds. Each column in a row is wrapped in a ShredThumbnail View
ShredThumbnail View	Represents a Shred thumbnail, consisting of a thumbnail image, and metadata about the Shred. Knows how to notify a listener if the Shred was clicked, in order to tell the listener to open a ShredModal View to play the Shred itself
ShredModalView	Represents the popup window that plays a Shred. Handles user events like rate button clicked and comment text submitted

Table 6.1: The set of Views that are implemented in Architecture 2.0

```
16
17 // In BaseBaseView.serialize:
18   5. Do nothing, expect this to be overridden
19
20 // In BaseView.postRender:
21   10. Do nothing, expect this to be overridden
22
23 // In concrete View.serialize:
24   5. populate and return a JSON object with proper Model and/or
       Collection data
25
26 // In a concrete View.postRender:
27   10. Fetch any necessary Models or Collections from the API
28   11. Render additional Html templates and inject these to the Dom
29   12. Create event listeners and map them to event handler
       functions
```

Note the number 5 and 10 indicates template functions that are implemented by the current subtype of BaseView. These are custom behaviors that cannot be implemented in the BaseView. All other operations like fetching a template, and rendering it however, are common behavior, and therefore implemented in the BaseView. This neatly shows the elegancy of the Template Method pattern.

The BaseView has one other necessary feature, which is to implement a clean-up function that is called every time a View is to be removed. Views are always removed whenever a new View is to be displayed instead of it, for example when the User goes to a new page inside the app, or clicks the next row of Shreds button. In these cases it is important to remove View's containing Html from the Dom tree, any also to remove event listeners the View created. Note that if a View contains child Views, then these Views must also be removed. Thus, the remove function in the BaseView is a recursive function that calls a clean-up method in every child View, if there are any, before it cleans up and removes itself when the recursion completes.

It is especially important to remove any event listeners when a View is deleted. If they are not removed, they will continue to exist and listen to events, so that if a View is recreated, its old events will co-exist with the newly created event listeners. Now, when an event is triggered, there might be multiple listeners listening to that event, and in affect calls to the same handler function, so that it is executed more then once. The result could be multiple equal write operations sent to the database. Also, this could lead to slow performance, because the listeners consumes memory.

Event handling

Each View is set up to listen to certain events that are relevant to that View. For example a ShredRowView is initialized to listen to the *next-row* button clicked, and a ShredModal View is initialized to listen to the *rate* button. In each View, there is an **event handler** function for every event it listens to. In Architecture 2.0 I have separated the types of events into two: **View-logic event** and **Domain-logic event**. A view-logic event is something

that simply manipulates the DOM tree in order to alter the Html. An example being when the User clicks the *next-row* button, in which case the responsible ShredRowView will respond by advancing the set of displayed Shreds to a fresh set of Shreds. A Domain-logic event however is more like a controller handler from Architecture 1.0, where an event that requires some business operation is triggered. An example is when the rate-button is clicked in a ShredModalView. This event requires some business logic, something that should not be implemented inside the View, as it would violate the separation of concern principle. Instead, the event is delegated to a particular function in a Model object, namely the a Shred Model object that represents the Shred that is currently displayed in the View. Often, however, a domain-logic event results in something that also requires some change in the user interface. Therefore, Domain-logic events will in certain cases perform view logic. However, this is implemented inside the View object, after the Model is done with the operation.

Html Templates

Each View has one or more Html templates that they are responsible for injecting into the Dom tree. Also, the Views know where in the Dom tree to put the particular Html template. For example the ShredRowView that represents the ShredRow of top-rated Shreds holds an Html template called *ShredsRow_topRated.html*, which the View will inject into the Html tag <*div id="topShreds"*></*div*>. Just like JSP template files in Architecture 1.0, the templates in Architecture 2.0 are not pure Html files, but contains special syntax that can reference Model data, and supports loop statements, conditional statements and other simple programming language statements. However, there is a big difference between the way I have implemented templates in Architecture 1.0 from Architecture 2.0. In Architecture 1.0, the templates where coarse grained, and contained a lot of view-logic to decide the outcome of the Html. In Architecture 2.0, I have decided to create many, and smaller fine-grained Html templates, and factorize out as much view logic as possible into the View. This is often done by letting Views have references to multiple fine-grained Html templates. These all have the advantage of being able to be reused in other parts of the app. Also, I have implemented a couple of fine-grained Views, that are being reused across the app. For example, a ShredThumbnailView is reused as a child View of other Views who need to display Shred thumbnails.

Abstracting View-logic out of the Html templates and into the Views, facilitates a better decoupling of Html markup and view-logic. This decoupling was not achieved in Architecture 1.0. One example: In Architecture 1.0, the ShredderView.Jsp contained many if-checks to figure out the relationship the User had with the particular Shredder that was to be displayed. A unique Html output was to be created depending on:

- if the visited Shredder is actually the same Shredder as the User
- else if the User has sent the Shredder a battle request

- else if a battle request from that Shredder is currently pending
- else if they are currently in a battle
- else; the User should then challenge the Shredder to a Battle

Therefore, the Jsp template had to include Html markup for every possible outcome, and depend on complex Jsp if-conditions to know which part of the Html to render (together with the rest of the Jsp page of course!). In Architecture 2.0, all of this is figured out **before** the rendering process begins. Now, the Html for displaying each of these five different shredder relationships are represented in separate (fine-grained) Html template files. This way, when the rendering process begins, the View will pick the proper Html template depending on the result of the if-check, and inject this template into the Dom. In effect, the templates contain very little view logic, only enough to display the data from a Model object it receives when the Html is rendered. This is somewhat easier to maintain in case the View becomes large and complex.

The Html templates are cached in the browser's JavaScript heap after the first time they are fetched from the Server. One problem however, with having many small Html templates is that it results in a large number of Http request-response cycles, because each Html template is a separate Http request sent to the server. A solution to this is to bundle all the Html templates into a single file, in a build process before the application is deployed. Unfortunately, I have not had the time to implement this for the thesis.

6.3.6 The Mediator

There are many cases in which disparate components need to communicate with each other in the app. For instance, separate Views need to communicate with each other, and sometimes Views need to communicate with Model objects they don't necessarily have direct references to. In order to facilitate a loosely coupled, flexible and efficient communication model, I have chosen to use the Mediator design pattern. This is a component where Views and Models can publish and subscribe to certain events, such that when someone publishes to the Mediator that an event has happened, the Mediator will notify every listening entity (subscriber), and call all of the handler functions the subscribers has registered with the Mediator. This solves the need to have many object references in every View and Model in order to call functions across the objects. The major events that are actively being used in the app are given below:

updateNavbar
: Called on the ScaffoldingView when something requires the top navigation bar to be updated. For example if the User adds a new fanee.

openUpShredModal
: Called on the Home, or Shredpool View (depending on who's act-

ive) when a ShredThumbnailView has been clicked, and a ShredModalView (pop-up window) is to open and display a Shred video.

authenticationFailed
Called on the HomeView when a User fails to login. Authentication is implemented in a separate module, which uses the Mediator to route authentication results to a proper handler (here, the HomeView) that will display an authentication failed message to the User.

authenticationSuccess
Same scenario as defined above.

acceptBattleRequest
Called on the Model object that implements the abstraction of a User. It will call a Rest operation on the API that creates a new battle between two Shredders.

addFaneeRelationship
Called on the the Model object that implements the abstraction of a User. It will register a new fanee relationship to the API.

6.3.7 Session

In Architecture 2.0, state is completely implemented on the client so that the server has no awareness of any logged-in User or session. In order to do this, the app needs to have a means of storing and manipulating state data on the client. This could be done by storing data in the browsers JavaScript memory, considering the single-page app scenario does not require the page to refresh. Unfortunately, this could negatively affect the browser's performance if the data size grows quite large, and also, if the User happens to manually refresh the page, the JavaScript memory is cleared. It could also be done by storing all the state inside cookies, but this is not as secure considering the state data needs to be transported in every Http request. This of course, would also waste and consume very much bandwidth. A solution is to use HTML5 WebStorage, which neither affects browser performance, or is subject to data loss on page refresh. The storage size is big enough to hold many megabytes of data (depends on the browser), so in practice there is need to limit how much User data to store in the browser. I have chosen to use session storage and not local storage, so that state data is restricted to a session. This is because the data I store in web storage is naturally bound to a "session", and shouldn't last for any longer then this. There is one misfortune with this design decision however; some old browsers do not implement HTML5 Web Storage. Now, I have not have the time to design a backup solution for such users, however a simple approach is to check during the bootstrap process if the current browser supports Web Storage, and if not, use the browser's JavaScript memory or cookies to store state data.

The app mostly uses sessionStorage to store User data only, considering much of the other state data is maintained in the Views (i.e JavaScript

memory). The storage is populated with User data when the User is authenticated. This data includes:

- User profile data, like username, address, birthdate, list of guitars etc
- Authentication details (a token made up of username and password)
- List of the User's fanees
- List of the User's current sent and pending battle requests
- List of the User's current battles

An API that wraps the session storage object is implemented in a separate module called Session (in session.js). This is a facade that offers handy functions like *getUser()*, *setUser()*, *getSentBattleRequests()*, *getBattles()*. Inside the facade, these functions manipulates the sessionStorage as if they were a database mapper. Considering sessionStorage only manipulates key-value pairs as text strings, the facade serializes JSON objects into a large structured string before it stores it in the sessionStorage data vault. Hence when the facade is to get data from sessionStorage, it will serialize the text string back to a JSON.

6.3.8 Summary of The Front-end

The App goes through a bootstrapping process when the client first accesses Shredhub. This process starts from a single script loading statement in the initial html page, that leads to the loading of the rest of the JavaScript application. After the App has completely initialized itself in the browser, it will start listening to Url changes in the browser. Every Url is hijacked by the router which will create a particular view instead of letting the browser send the request to the server.

Models and Collections implement the domain of the application, and is built with the domain model pattern, and the active record pattern.

Views represent a particular part of Html, and is responsible for listening to and handle events that occur in their owning Html. Views have references to Models and/or Collections. Views can be nested in order to delegate complex and coherent View-logic into sub Views.

Independent modules communicate through a central Mediator in order to avoid complex references and to provide controlled message routing.

The App implements the notion of a session by using the browser's sessionStorage to persist session data. Together with state data that is kept in the JavaScript Views, the App completely maintains all of the application's state.

6.4 The Back-end

6.4.1 The Rest API

The Rest API is the communication boundary between clients and the server. Models and Collections in the App communicates with the backend through the Rest API in order to perform CRUD operations on themselves, in the database. The back-end exposes all of its available operations through the Rest interface. The operations are resource-oriented, meaning they are centered around the application's domain. The domains are *Shreds, Shredders, Battles* and *BattleRequests*. For each domain, there are four main operations that are offered, one for each Http method: *Get, Post, Put* and *Delete*. Now, in order to offer more complex operations then just a combination of a resource and an Http method (e.g Get + Shred), the Rest API adds an additional verb that describes a specific operation that is to be performed. One example is *Get + Shred + bestRated* which fetches the best rated Shreds on Shredhub. Given below is a list of some central operations exposed in the Rest API. These should be fairly self-explanatory:

Shreds

- GET: api/shreds/NewShredsFromFanees/<uid>/?offest=<o>&page=<p>
- GET: /api/shreds/shredsByTags/?tags=<t>&offest=<o>&page=<p>
- POST: /api/shreds
  - Request body: JSON {Shred}
- PUT: /api/shreds/<uid>

Battles

- POST: /api/battles
  - Request body: JSON {Battle}
- GET: /api/battles/battlesForTwo
  - Request body: JSON {Shredder1, Shredder2}

BattleRequests

- DELETE: /api/battleRequests/<uid>
- GET: /api/battleRequests/shredder/<uid>

Shredders

- GET: /api/shredders/mightKnowShredders/<uid>/?offest=<o>&page=<p>
- GET: /api/shredders/<uid>/addFanee
  - Request body: JSON {Shredder}
- GET /api/shredders/<uid>

And in addition there is one central operation:

- /api/authenticate
  - Request body: JSON {Base64 encoded username:password}

The <*uid*> field is a unique identifier for the resource. In addition to the resources and verbs that define an operation, many of the API offer support for additional arguments in the query string and in the request body. Also, all of these Restful Url's must contain an authentication token that the backend uses to verify that the User is allowed to perform the operation. The App appends this token to the Http Authorization header parameter on every API request.

It's important to notice that these URL's are all self-contained, in that they have all the information needed to perform the operation. Take for example the URL *GET: api/shreds/NewShredsFromFanees/5142b8fc174328d087ac49b9/?offest=20&page=3*. The long string represents a unique Id (uid) for a Shredder. With this request the backend will query the database for a set of Shreds that are made by the Shredder with the given uid's fanees. Also, for security reasons, the backend checks that the uid matches the User identified in the authentication token. This scenario is implemented in many of the API operations where the backend must be sure the calling User is allowed to perform a particular operation. The returned list is a set from the query result, starting at result number 3*20, and the size of the result being 20 Shreds. In a similar operation in Architecture 1.0, the back-end would look at the Http Session to find the User (i.e a Shredder) who issued the request, and by knowing what page number the User is currently at, and the amount of Shreds that are displayed on the current page, the back-end would have all necessary information to issue the request. Hence all the necessary information in that case is on the server.

In cases where complex data structures need to be stored or updated in the database, the client sends this data as JSON objects in the Http request's response body. An example of a Shred that is saved to the database could look like this:

```
1
2 Request URL: http://localhost:3000/api/shreds
3 Request Method:POST
4 Content-Type: application/json
5 Request Payload
6 {"description":"Sweet Shred in C-minor",
```

```
7  "shredRating":
8  {
9    "numberOfRaters":0,
10   "currentRating":0
11 },
12 "shredComments":[],
13 "owner":
14 {
15   "_id":"5142b8fc174328d087ac49b9",
16   "username":"Michael"
17 },
18 "tags": ["Scale","Speed-picking","Melodic"],
19 "shredType":"normal",
20 "timeCreated":"2013-03-18T12:24:13.363Z",
21 }
```

In this Rest operation, the backend responds with a status code, indicating if all went well, in addition to a JSON object, being the Shred after it is saved to the database.

The back-end always return an Http status code which serves to inform the client if the operation was successfully executed or not. The Http status codes used are:

- 200 OK, meaning the operation was performed, and the response contains JSON data
- 401 Unauthorized, meaning the User is not allowed to issue this request. An example is if the User tries to add a Shred, and the owner is set to reference a Shredder who's un-equal to the Shredder identified in the authentication header.
- 400 Bad Request, meaning the User tries to perform an operation with illegal input parameters. An example is if the User tries to add a rating to a Shred with a value higher then 10.

There are many other status codes supported by Http, which I could have used in order to enrich the error messages used in the application. However, this goes a bit out of scope for this thesis. The point here is to show how error handling can be done in a stateless and decoupled fashion; the back-end does not know how the App treats the error message. This is apposed to Architecture 1.0, where in cases of an error, the server will return a completely rendered error page back to the client.

6.4.2 The Data Repository Layer

The data repository layer is the part of the backend that implements the Rest API and communicates directly with the database. It is organized as a set of controller modules; one for each domain resource. This is JavaScript code that runs on a Node js server. Much like controllers in Architecture 1.0, the controllers in Architecture 2.0 are mapped to a specific (Restful) Url. However, instead of going through a complex domain logic layer, and data source layer, the controller's responsibility is much more simplified. Generally a controller handler does:

1. Validate the parameters given in the Url query string, request body and authentication header.
2. If there is illegal input, send a proper Http status code back to the client.
3. If not, create a database query with the Url arguments and execute the query on the database.
4. Send the result (with no modification) back to the client.

Notice the last statement. This is because the data format returned from the database happens to be JSON, which is the one transmission format used by the Rest API, and also the App. This point highlights an important attribute of Architecture 2.0, namely the simplicity of the backend architecture. There are other popular transmission formats that can be used as well, for example XML. However, this format is somewhat more complex, and requires marshaling considering it is not a data format supported natively in JavaScript.

6.4.3 Authentication

Authentication in Architecture 2.0 is implemented with the Http basic authentication protocol. The App authenticates Users through the Rest API by concatenating the User's username and password into a base64 encoded string. This string is appended to the Http authentication header parameter, and is sent with every API operation (except the initial request for the home page). The reason it is sent with every request, is in respect to *Reference-model 2.0*, where the server is stateless. Hence, every Url request must contain User information. Now, Http basic authentication is not a complex and especially secure protocol, so the solution is not optimal. A first improvement is to enforce the use of Https in order to properly encrypt the the username and passwords. Other authentication protocols could also have been chosen. One popular solution is OAuth, which is much used in Web 2.0 applications. However, this is a somewhat complex protocol that requires some effort to implement. This is why I decided to go for something simpler, that still conforms to a stateless solution.

6.4.4 The Databases

There are two databases used in Architecture 2.0. The reason for this is because I have two different persistency needs. One is to persist the domain model in a flexible and efficient way, which is done with MongoDb. The other is to have authentication data available in a highly efficient manner, which is done with Redis.

User Authentication with Redis

In Architecture 2.0, the authentication token needs to be verified in every Url request except those regarding the home page. Therefore, the back-

end must have a highly efficient way to authenticate an API request. The solution I have made for this is to store authentication details in Redis. With Redis, I store two key-value pairs for each User, one that maps a username to a unique Id, and the other maps the unique Id to the password that belongs to that User. The unique Id is the same unique Id that is used for that particular user in MongoDb. An example of a User in Redis looks like this (The long string represents a unique identifier):

```
username:Michael:uid 5142b8fc174328d087ac49b9
uid:5142b8fc174328d087ac49b9:password 1234
```

Keys are on the left-hand side of the white space, while values are on the right. The colons are used to infer a descriptive semantic. For example the key *username:michael:uid* neatly describes the value *unique id for an entity with username equal to "Michael"*. A similar semantic applies to the second key-value pair. In order to authenticate a User, the backend does the following lookup:

```
1
2  function authenticateUser(username, password) {
3
4    // Create a string on the form ''username:<username>:uid'':
5    usernameStr = ''username:'' + username + ''uid''
6    get the value with key=usernameStr from Redis, put result in res
7
8    if ( success ) {
9    // A user exists with the given username. Now check the password
10     // Create a string on the form ''uid:<uid>:password''
11   var uid = res.toString();
12   var passwordStr = "uid:"+uid + ":password"
13   get the value with key=passwordStr from Redis, put result in res
14
15   if ( success ) {
16    if ( password === res.toString() ) {
17    // Correct password was given. Return success together with the
          uid
18    }
19   }
20   // Authentication failed, return proper error message
21 }
```

The uid is returned so that it can be used to fetch the newly authenticated Shredder from the database.

The reason Redis was chosen is because of its extremely high speed when it comes to simple key-value pair lookups. Redis is not meant for complex and structured data, but is specialized to operate on simple HashMap data structures. Also it favors speed over durability, something that is preferable in this occasion, considering the only time I perform write operations to Redis is when new Shredders are created. If the system was to crash however, before any newly created Shredders are saved to disk, it wouldn't be that big of an issues, considering all the data except the password is already stored in MongoDb. It would just be a matter of asking the User to create a password.

MongoDb

The domain in Architecture 2.0 is persisted using MongoDb. The reason I chose MongoDb for this, is mainly because it uses a JSON-like format to persist data, which is a very nice fit for the domain; much of the domain has a nested structure, which is very appropriate to implement with JSON. In general, this nested data structure is very typical Web 2.0 apps that has blog-posts and comments (with commenters). Also, considering the MongoDb database can be manipulated directly using JavaScript, there is no need to implement additional data mappers for creating queries and serializing query results. A final reason I chose MongoDb is because of MongoDb's schema-less document model, allows for highly flexible data modeling solution. Hence, I can very easily customize my MongoDb collections to fit the data exactly like they are displayed in the App. This does require some duplication of data, but avoids tedious relations across collections, that normally requires join operations in order to fetch the necessary data.

Examples of the set of MongoDb collections implemented i Architecture 2.0 is given below:

Shredder

```
"_id" : ObjectId("5142b8fc174328d087ac49f7"),
"username" : "Shreddaz64",
"fanees" : [
    {
      "_id" : ObjectId("5142b8fc174328d087ac49f5"),
      "username" : "Shreddaz62",
      "profileImagePath" : "EddieVanHalen.jpg"
    }
],
"birthdate" : ISODate("2013-03-15T06:00:28.202Z"),
"country" : "Denmark",
"profileImagePath" : "Hslash.jpg",
"email" : "shredder64@slash.com",
"guitars" : [
  "Gibson flying v"
],
"equiptment" : [
  "Orange"
],
"description" : "Simple test shredder #64",
"timeCreated" : ISODate("2013-03-15T06:00:28.202Z"),
"shredderLevel" : 84
```

Shred

```
"_id" : ObjectId("5142b90a174328d087ac4a2e"),
"description" : "Simple test shred #19",
"owner" : {
  "_id" : ObjectId("5142b8fc174328d087ac49c4"),
  "username" : "Shreddaz13",
  "imgPath" : ''ShreddazImage.jpeg''
},
"timeCreated" : ISODate("2013-03-15T06:00:42.313Z"),
"shredType" : "normal",
```

```
"shredComments" : [
  {
    "timeCreated" : ISODate("2013-03-15T06:00:42.313Z"),
    "text" : "Lorem ipsum lol cat mode19",
    "commenterId" : ObjectId("5142b8fc174328d087ac49ea"),
    "commenterName" : "Shreddaz51"
  },
],
"shredRating" : {
  "numberOfRaters" : 946,
  "currentRating" : 8188
},
"videoPath" : "battle-23-12-4.mp4",
"videoThumbnail" : "battle-23-12-4.jpg",
"tags" : [
  "Sound test",
  "Pop"
]
```

Similarly there are collections for Battles and BattleRequests. Notice there are only 4 different collections in this MongoDb implementations. This can be compared with the SQL implementation from Architecture 1.0 that is implemented with 16 tables. The reason I have chosen to limit the amount of collections as much as possible is to avoid the tedious join operations that would normally be needed in SQL. Joins are very slow, and also, they are not natively supported in MongoDb. One has to manually implement joins by performing multiple subsequent read operations across documents. My solution however emphasizes the use of duplicating data so that documents fit the domain in the way they are visualized in the App. Look for example at the Shred document in the example above. The owner consists of his Id, username and image path. Also, the comments contain the comment-owner's Id and username. This is exactly enough data that is necessary in the App, in order to visualize the Shred. In a normalized SQL (i.e Architecture 1.0) implementation the owner would just be represented by a foreign key, and during a Shred-fetch a join operation would have to be done for the Shred-owner, all the comment owners, every tag, and every rating. One misfortune with this design, however, is if any of the duplicated values were to change in the original document. For example if the Shredder with name Shreddaz13 was to change his profile image. In this case this update would have to be propagated to every place in the database where that particular image is referenced. However, I have acknowledged this fact simply because profile images aren't something that is likely to change very often. Another misfortune is that the database is somewhat App-aware. Imagine the API was to be used by other clients, maybe third party clients that would have other requirements to the amount of data that is populated with a particular fetch operation. One could argue that this customized duplication of data is somewhat enclosed for future needs. However, I have acknowledged that this data modeling decision is still very flexible, considering every domain resource always include their uid, making it possible to force join operations if more data needs to be populated in a given query.

6.4.5 Summary of The Back-end

The back-end is built as a Rest API that exposes a set of public operations. These operations are completely self-contained in that they contain all the information that is necessary to perform the operation without relying on any previous execution. Hence the back-end is completely stateless.

Authentication is implemented with the Http basic authentication protocol. In order to no rely on a session implementation on the server, every Rest operation contains authentication details. This does require some amount of extra processing on the server during each request. However, every User's authentication details are stored in a Redis database, making lookups highly efficient.

MongoDb was chosen to persist the domain objects, because MongoDb stores data in a JSON-like format, which is a very nicely fits the domain's data structure. The database is highly flexible, making it easy to store objects exactly how they would look like in the App. This is done by favoring duplication of data over structured relations that requires join operations.

6.5 Summary

In this chapter we have looked at Architecture 2.0. This architecture is very much different from that of Architecture 1.0. Here, we have completely abandoned the state-full thick server by letting the application resolve around a large-scale JavaScript application that completely runs in the client. The server's job is to serve this application to the client's browser during the initial Shredhub request, a process we called the bootstrap process.

The front-end application has a decoupled structure where the code is organized into coherent modules. Domain logic is implemented in Models and Collections, while state and view-logic is implemented in Views. Alto, there is a central Router that hijacks Url requests to avoid Urls being sent to the server.

The back-end is built around a Rest API that exposes self-contained operations to client users. The API uses JSON as the data format, which naturally fits into the overall programming environment, that is purely JavaScript based. Authentication does not rely on sessions on the server, because every Rest request contains authentication data. This requires fast authentication on the server, something that is solved by using Redis as a User/authentication database. The domain model is persisted with MongoDB where I have favored a nested and somewhat duplicated data modeling scheme, in favor for complex relations that requires join operations. The result is that the data in the database is highly optimized for the application's needs, but a potential drawback is that the data is somewhat rigid.

Chapter 7

Performance and Source Code Analysis

A major goal for any interactive Web application is to minimize the response time for user actions, because it has a significant impact on the user-experience. Users most often don't have patience to sit around and wait for slow page requests, which in some cases could result in Users abandoning the site in favor of other competitors. The response time for such an action is a measurement of the time it takes from when the User initiates the action, until the result is completely visible in the browser. An action might be that the User clicks a button, follows a hyperlink, presses the enter key in a search field etc.

The response time depends on many performance factors in the Web app. In general, these are:

Application server's throughput , which concerns how many requests the application server can handle per time unit.

Database server's throughput , which concerns how many transactions the database can handle per time unit

Client-tier efficiency which concerns how fast the browser can render and display the result of a User-action. Often this depends on the JavaScript implementation that is required to display the result.

There are also other factors that affects the response time of a web-app, like network performance and the hardware that hosts the client and the server. However, these issues will not be considered in this thesis, mostly because performance tuning these elements does not directly indicate any pros or cons in the two Web architectures that are studied in this thesis.

The application server's performance also depends on the scalability of the system. Scalability is a measurement of resilience under ever-increasing load. Hence another goal is to maintain the performance levels when the number of concurrent users increases. Scalability also depends on the application server's throughput, and the database server's throughput.

Now, in order to analyze and compare performance and scalability properties for the two architectures, a number of system tests have been

designed and executed in real deployment scenarios. Equal tests have been run on both prototypes. Now, one important thing to mention is that the application server's throughput also depends on the web framework that is used. So considering the fact that the architectures uses two different frameworks (Node Js and Java Swing), might cause subjective results for comparisons. However, I was aware of this fact before I chose to use two different web frameworks, and therefore I built Architecture 2.0 with Spring as well (not really a big effort, just a matter of implementing the API in Java). The performance results for Architecture 2.0 on Spring vs Architecture 2.0 were very similar, and therefore I chose to do all the testing based on the Node Js version.

A final goal for this project is to analyze and compare the source code for the two architectures. The source code is to be measured in terms of code flexibility and maintainability, which is an important property in order to facilitate future code modifications and extensions. Now, as mentioned in the introduction chapter, such a comparison is a difficult and laborious task. Therefore the analysis of the source codes is somewhat short. However, a proper test case has been designed and implemented on both architectures, and the results are relevant. We will look at this in the end of the chapter.

7.1 Hardware and software used for testing

Both prototypes have been deployed on a test machine stationed in Madrid, Spain. The machine was hosted by PlanetLab, a global network of computers made available for researchers to develop, deploy and test distributed systems. The reason I chose to deploy it in Spain was in order to get realistic transmission times. The system specifications for the test machine are as follows:

Operation System	Fedora 12
CPU Architecture	Intel(R) Core(TM)2 Quad CPU Q8400
CPU Clock Speed	2.66GHz
CPU Cache Size	2048 KB

Performance testing was made with the Chrome Developer Tools, [**chrome**] which is a browser feature for the Chrome browser that captures Url requests, and monitors JavaScript executions. The tool not only calculates the complete response time, but also gives a detailed overview of the times spent for each individual Web resource that is fetched from the server. Both Mozilla Firefox and Internet Explorer have similar tools for web performance testing and profiling, and they mostly deliver the same functionality. I chose to use Chrome simply because I am familiar with it, and it doesn't require any extra plugin installation. Stress testing the server was done with Apache JMeter, which is a Java program that is able to execute and monitor multiple threads. These threads can be configured to do Http requests.

7.2 Performance and Scalability Tests

In this section, we look at the concrete tests that have been made, and the results of these. The tests are separated in five different sections. The first two tests investigate the performance of the prototypes by inspecting request response times, the third test investigates the scalability characteristics of the prototypes by stress testing the back-end implementations, the fourth investigates database speed, and the last test investigates the architectures' source code in terms of flexibility and maintainability. What is missing here, is a test case that investigates the database's throughput properties. Unfortunately I did not have time to create this test.

For testing purposes, an equal set of dummy objects were created in the two databases. These are:

- 1000 Shredders
- For every Shredder, a number from 0 til 10 fanees
- 100000 Shreds
- 100000 Battles
- 10000 Battle requests

7.2.1 Test 1 - Page Loading Tests

Goal: To determine how fast the prototypes create the major pages of Shredhub. Also, to investigate the times spent in the various phases during a request
Tests: Response time/round-trip time
The page loading test is meant to investigate the amount of time the User has to wait from the time a Url is requested, until the page is displayed in the browser. This test is important in order to identify which of the two prototypes are capable of creating a page the fastest. In addition to calculating the response times, the test also performs profiling of the various phases of the page loading processes, in order to identify how much time is spent on the server, and how much time is spent preparing the page in the browser.

The page loading test was performed actively by a human user. The tester would either, for the initial page, write the url in the address bar of a Web browser and press the enter button, otherwise, the tester would click on a link inside Shredhub that leads to the given page. I have chosen to use the Chrome web browser to run the tests, because it comes with the Chrome developer tools. Every test was done 5 times, in which the test results show the average of these.

The result figures show the complete round trip times for all the different server requests. These results are displayed in waterfall figures, which naturally depict the ordering of the different resource fetches. The waterfall models also capture the fact that some resources depend on each other in order to start fetching. For example, the browser will start fetch

CSS/JS resources as soon as it has gotten the Html page from the server. For every test result, there is a number saying how much time was spent on the server, how much data was sent from the server (without images), and when it was fair to say that the page was visible in the browser. The latter timing represents the time when everything but the images were completely rendered. This is because rendering images is the last thing the browser does (in the case for Shredhub), and it says nothing about the performance differences for the two prototypes. The timings in the result figures are displayed in milliseconds. For Architecture 2.0, I have chosen to indicate the amount of time spent doing Ajax requests to the server (the timings concerns the time it took on the server + Http transmission times), as well as showing the time of the last JavaScript execution in the App. The last JavaScript execution indicates the time when the App has rendered every necessary Html template and written this to the Dom. Hence this is when I acknowledge that the page is finished and displayed in the browser. As for Architecture 1.0, the first request in the waterfall model always represents the time it took to process the request on the server plus the Http transmission times regarding this. I acknowledge that the page is finished and displayed in the browser when the DomContentLoad event is triggered, because this is when the browser has rendered every element in the Html page.

Note that the tests were performed on a somewhat slow network connection. This explains the slow timings, especially for image resources. However, because the testing on the two prototypes was done on the same network connection, the network overhead isn't relevant for the test results.

The Urls that have been tested are:

1. *www.shredhub.com/*
2. *www.shredhub.com/theshredpool*
3. *www.shredhub.com/shredders*
4. *www.shredhub.com/shredder/1234*

These are the four main pages of Shredhub. They all require database lookups in order for the pages to be generated. All of the pages except the first one requires the User to be logged in. In this case I have actively logged the User in before the real page loading tests were made.

Figure 7.1 on the next page shows the results for loading www.shredhub.com. Architecture 2.0 is very much slower then Architecture 1.0. The reason is that Architecture 2.0 has to load a big pile of JavaScript (the whole app!) before the server can start to render the real page. This is clear from the waterfall figure; once the App has loaded, the browser will start executing the JavaScript statements in the App, which will create templates and execute an Ajax request for Shreds in the database. On the other hand, Architecture 1.0 spends more time processing on the server, and the User has to wait 532 milliseconds before he can see anything at all on the screen. An advantage with Architecture 2.0 here, is that the User can see a minor part of the page already after 249 milliseconds. But this is just the scaffolding Html that

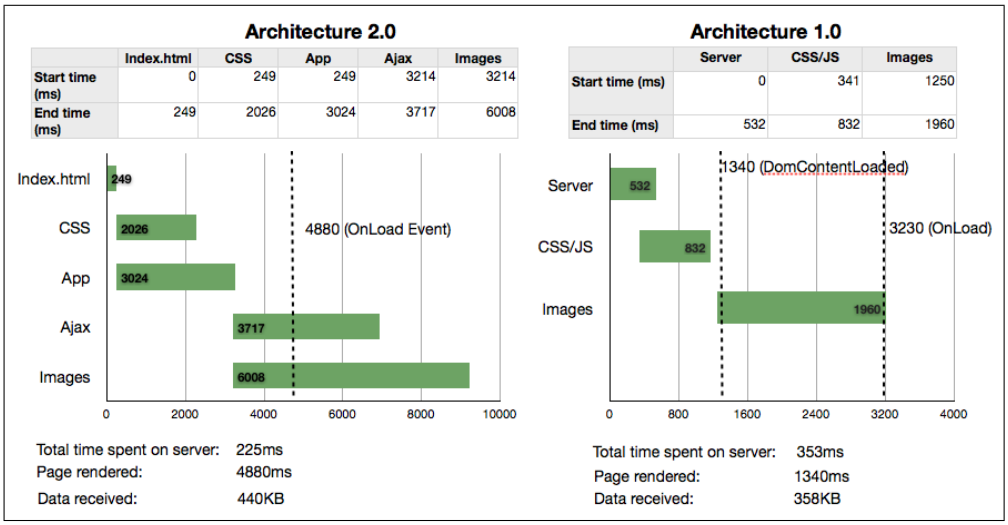

Figure 7.1: Test results for www.shredhub.com

is contained in Index.html, which is basically just a "Shredhub" headline. However, it does give the User something to look at much quicker then for Architecture 1.0. A final notice is that Architecture 2.0 sends more data to the browser. This is primarily because it sends the whole App.

Figure 7.2 on the following page shows the results for loading www.shredhub.com/shredpool. Architecture 2.0 is displayed a little bit faster in the browser then Architecture 1.0, however the reason it is somewhat slow is because it has to perform 8 Ajax requests. The browser executes these in parallel (hence the 2180/8 ms for time spent on server), but it is still very time consuming. Architecture 1.0 is slow because it has to perform an Http redirect after authenticating the User. The real work happens during the work on rendering the Shredpool on the server. Altogether, Architecture 1.0 spends less time on the server, but is slower because of the redirect. Also, Architecture 1.0 sends more data. This is because the Shredpool html file fully rendered is quite big.

Figure 7.3 on the next page shows the result for loading www.shredhub.com/shredders. Architecture 2.0 is almost twice as fast as Architecture 1.0. This is because Architecture 2.0 only fetches a small set of JSON Shredders from the server, and executes only a little bit of JavaScript in order to render the new page in the browser. As for Architecture 1.0, even though the execution on the server is quite fast, the result shows up late in the browser because the page that is sent is quite big and it is time consuming for the browser to render the whole page. Again, Architecture 2.0 sends much less data to the browser then Architecture 1.0. It's just an array of 20 JSON Shredders.

Figure 7.4 on page 129 shows the results for loading www.shredhub.com/shredder/<uid>. Again, Architecture 2.0 scores better, and the reasons mostly the same as in the previous test. The App merely has to perform a tiny bit of JavaScript in order to create the new page, and there is just one Ajax call to the server in order to fetch the necessary data. Notice however that the amount of time

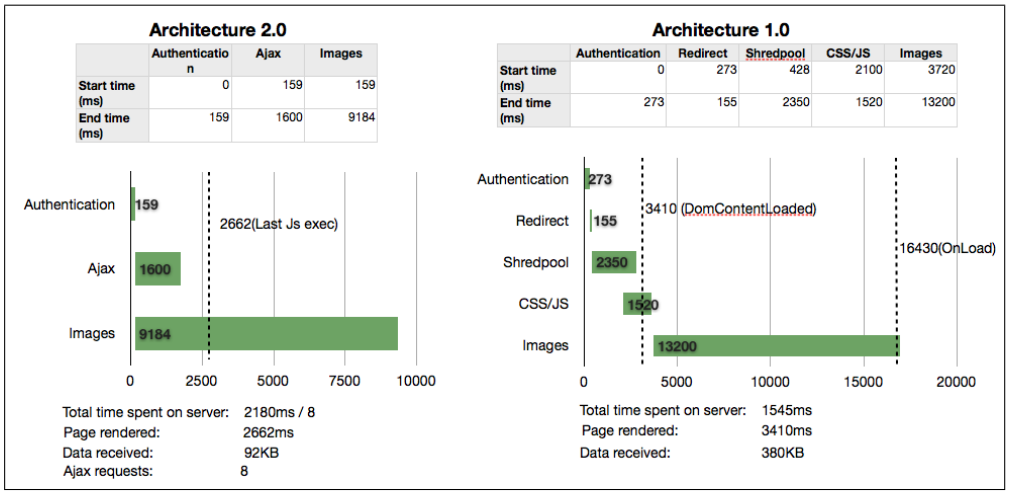

Figure 7.2: Test results for www.shredhub.com/shredpool

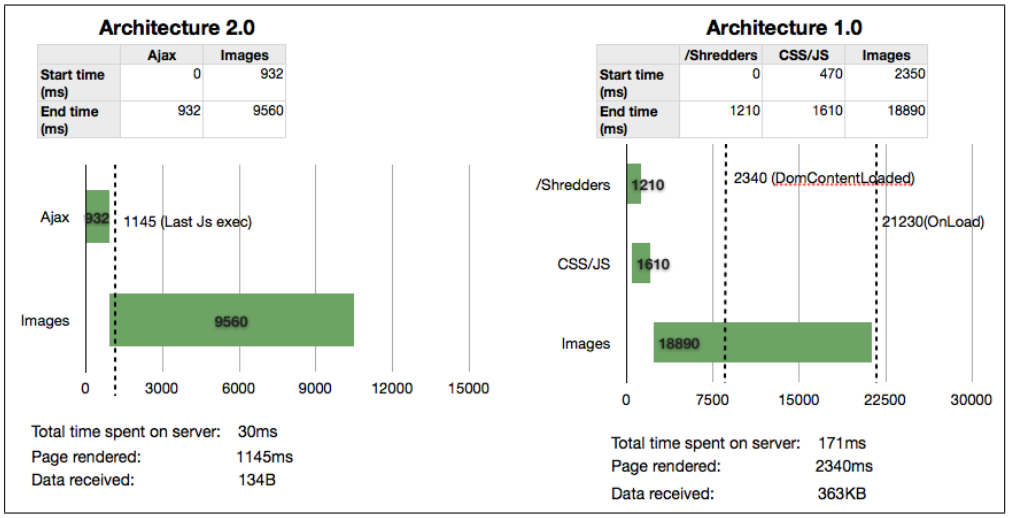

Figure 7.3: Test results for www.shredhub.com/shredders

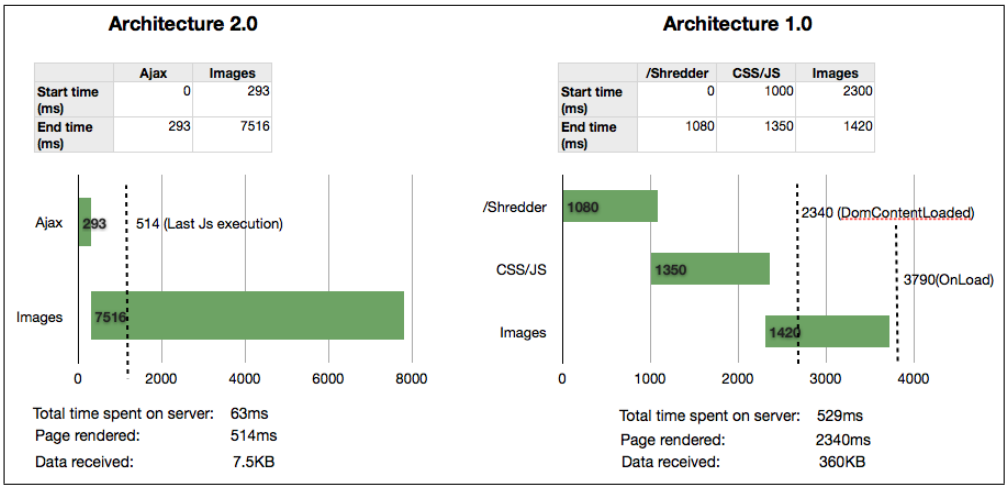

Figure 7.4: Test results for www.shredhub.com/shredder<uid>

it takes after the Ajax request has finished and the last JavaScript execution. This time difference is bigger then in the previous test, because the App has to execute some extra business operations in order to figure out the relationship between the visited Shredder, and the logged in User. Architecture 1.0 on the other hand is slow for the same reasons as in the previous example; the page that is created is big, and the browser has to render it from the ground up once it is received from the server. Also, like Architecture 2.0, Architecture 1.0 spends some time performing business operations before it starts rendering the Jsp page and sends it to the browser. It also sends more data to the browser, yet again because of the large html file.

7.2.2 Test 2 - Interactive User-Action Tests

Goal: Determine the response time for interactive user-actions on Shredhub
Tests: Response time/round-trip time
This test is investigating the response time spent when a User performs a particular interactive task on Shredhub. Unlike test 1, which is investigating response times when complete Web pages are requested, this test only concerns minor interactive actions that happens inside a page, possibly without any server involvement. Again, the results are showed in waterfall models that capture the various phases for each request. The results also shows the time spent on the server, the time the end result was displayed in the browser, and the amount of data sent to the browser.

The user-actions tested are:

1. *The User clicks next on a shred row*
2. *The User clicks on a Shred that opens a new video window*
3. *The User comments a shred*
4. *The User rates a shred*

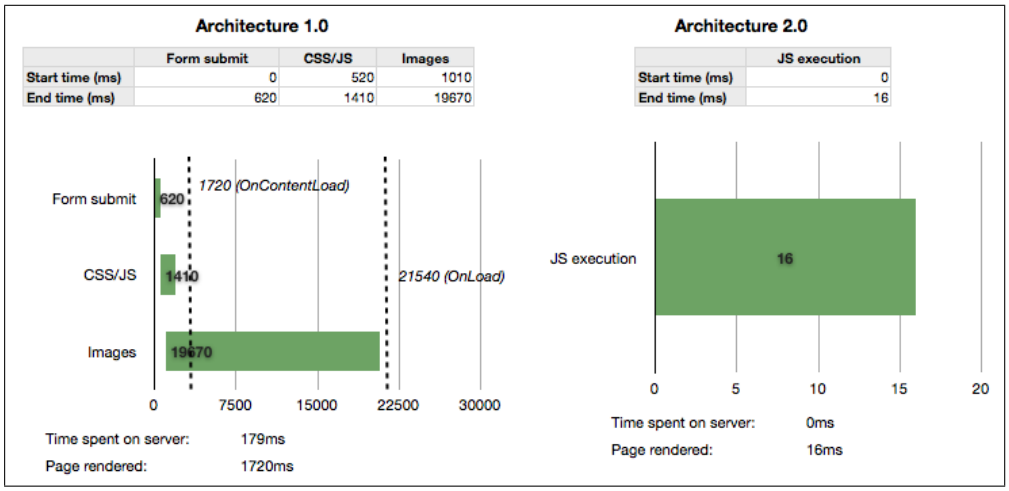

Figure 7.5: Test results for when the User clicks on next shred row

Just like in Test 1, the Chrome development tool to investigate the round-trip time for each action.

Figure 7.5 shows the result for User-action 1. Architecture 2.0 is very much faster in this case. The reason is that it doesn't have to consult the server; the next set of Shreds was fetched when the Shredpool was accessed, so it lives in the browser's JavaScript heap memory. It only has to execute some JavaScript code in order to alter the Dom to show the next row of Shreds. Architecture 1.0 is slower because it has to make an Http request to the server. Even though the next set of Shreds are cached in the session, the Http round-trip adds to the cost of displaying the page. Notice also that Architecture 1.0 does a page refresh here, because it is a form request, not Ajax. That explains why so much data is sent from the server.

Figure 7.6 on the facing page shows the result for User-action 2. The results are basically the same as in the previous test. The only difference is now Architecture 1.0 spends even more time on the server, because it has to fetch the Shred from the database.

Figure 7.7 on the next page shows the result for User-action 3. In this case the results are fairly equal. However Architecture 1.0 is a tiny bit faster, simply because it doesn't execute as many JavaScript statements as Architecture 2.0. The reason for this is that the JavaScript in Architecture 1.0 are simple self-contained handler functions of less then 10 lines of code. The same functionality in Architecture 2.0 is implemented as part of a bigger code base, and has to go through several function calls and object instantiations in order to execute the action. Also, note that the times spent on the server are fairly equal for both prototypes. This makes sense considering that there is no business operations performed for Architecture 2.0, and Architecture 1.0 performs just a tiny amount of validation operations.

Figure 7.8 on page 132 shows the result for User-action 4. The results

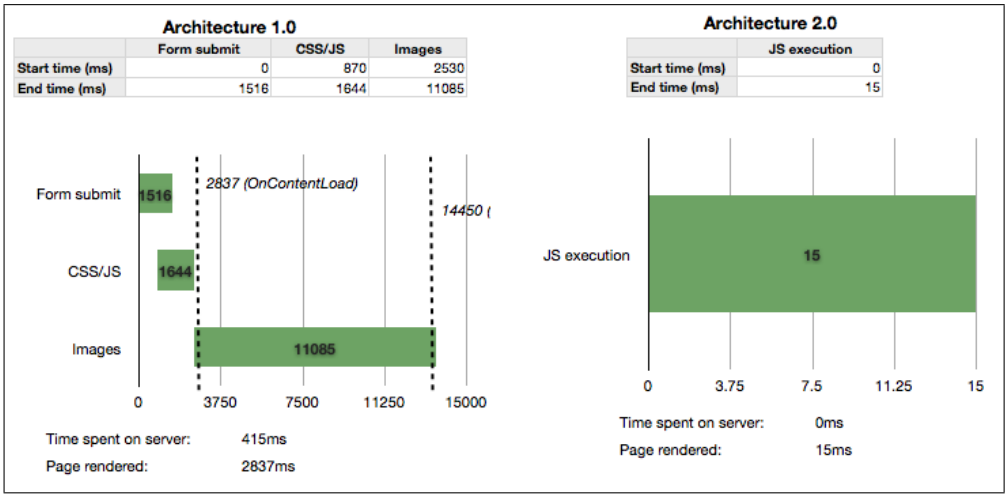

Figure 7.6: Test results for when the User opens a Shred window

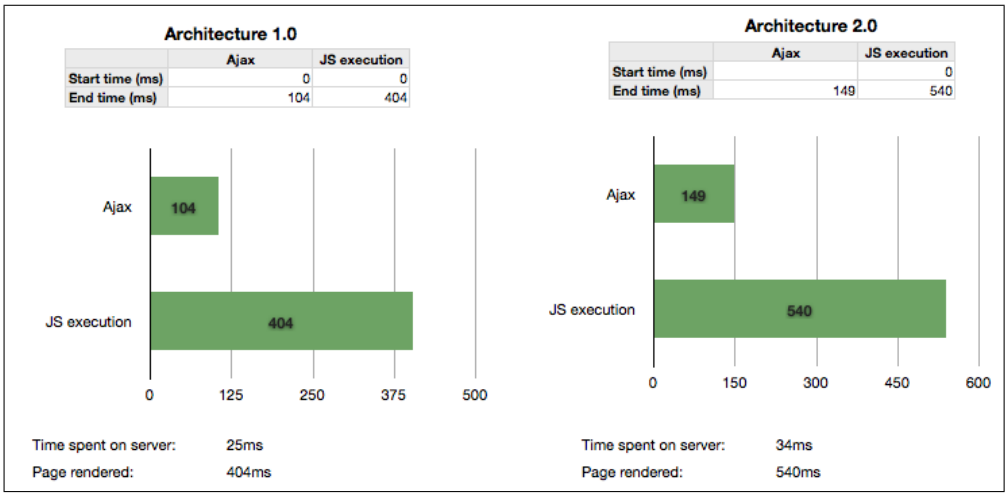

Figure 7.7: Test results for when a User comments a Shred

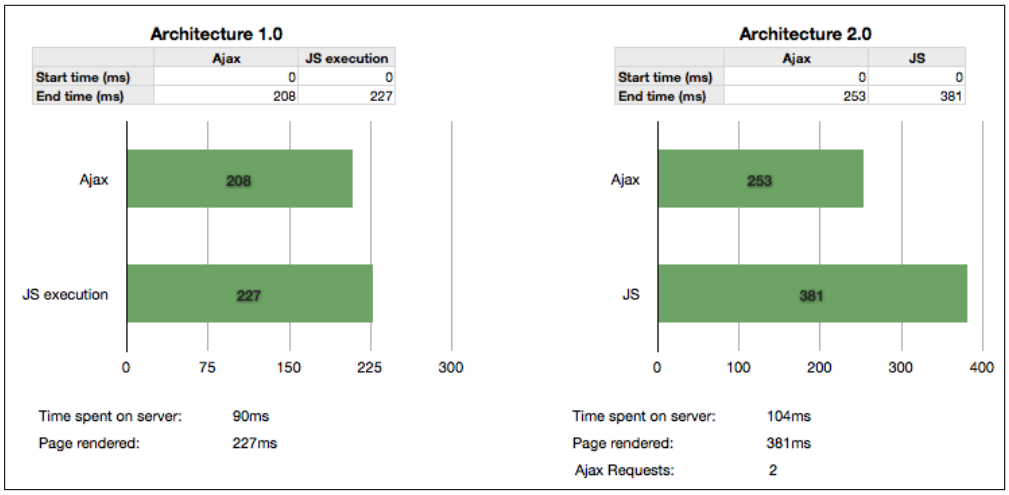

Figure 7.8: Test results for when a User rates a Shred

here are fairly equal to the previous test. Architecture 1.0 scores a little bit better because there is a lot less JavaScript to execute in order to handle the request.

7.2.3 Test 3 - Back-end Scalability Test

Goal: Determine how many concurrent requests the prototypes can support, and how quick the server handles requests under heavy load.
Tests: Scalability of the back-end implementations
This test was performed by creating multiple threads that executes a set of predefined actions on Shredhub. The actions are meant to simulate a normal flow of user-actions, to get a best-as-possible view of how well the server scales under common user-scenarios. The tests were created and executed with Apache JMeter. This was configured to have one test case that issues many subsequent actions:

1. *The User visits the home page*
2. *The User logs in and visits the Shredpool*
3. *The User uploads a shred*
4. *The User watches a shred*
5. *The User comments a shred*
6. *The User accesses the page www.shredhub.com/shredders*
7. *The User clicks on a particular shredder, which leads to the page www.shredhub.com/shredders/<uid>*

Each thread (that is, a User) executes all these actions on the server, once. In order to perform stress testing, JMeter is set up to generate an

Users	Throughput	Avg Request Time	Error
1	0.05	3.7	0
5	0.25	4.1	0
10	0.58	2.7	0
50	1.96	2.7	0
100	2.88	3.5	0
200	3.2	15.3	0
400	2.06	59.5	0
600	1.97	99.8	0.05

Table 7.1: Load testing results for Architecture 1.0

Users	Throughput	Avg Request Time	Error
1	0.06	0.6	0
5	0.27	0.4	0
10	0.53	0.43	0
50	2.27	1.45	0
100	3.95	2.5	0
200	3.8	3.3	0
400	5.9	9.7	0
600	4.54	15.1	0
800	6.38	16.3	0
1000	5.21	17.1	0.07

Table 7.2: Load testing results for Architecture 2.0

increasing amount of simultaneous threads until the server starts to return erroneous responds. Now, JMeter needs a given amount of time in order to be able to create enough threads without saturating the test computer. This is called the ramp-up time. JMeter was configured to create the number of threads T with a ramp-up time = N seconds, where N = T for values of T from 1 to 100. All subsequent amount of threads T will be created with a ramp-up time of 100 seconds. The ramp-up period was configured this way in order not to create an unusual high hit rate on the server, which would be an undesirable behavior. The client test machines weren't able to issue more then 300 simultaneous threads executing the test case. Therefore, I had to add a new test machine for every n*300 threads. The results are showed in tables 7.2.3 and 7.2.3.

Users = Number of concurrent threads

Throughput = Average requests per second

Avg Request Time = Average seconds spent per Http request

Error = Average percentage of requests with error

Query	CRUD	Architecture 1.0	Architecture 2.0
ShredsByRating	R	244ms	241ms
ShredsByFanees	R	11ms	68ms
ShredsByFaneesOfFanees	R	202ms	397ms
ShredsByTags	R	212ms	106ms
CreateShred	C	64ms	40ms
AddShredRating	U	27ms	16ms
DeleteShredComment	D	33ms	11ms

Table 7.3: CRUD

The results show that Architecture 2.0 can handle more concurrent Users then Architecture 1.0. In addition, Architecture 2.0 has relatively quick request times compared to Architecture 1.0. A misfortune with Architecture 1.0 is that after 100 Users, the request times were very high. The reason for this is mainly because of the high memory consumption on the server when the number of active Users is high. The server can handle them, but the processing time is slow. I chose to stop increasing users after respectively 600 and 1000, because after this, the error percentage increased drastically. A final thing to mention that is not depicted in the table, is that PostgreSQL applies caching such that for read queries, the fetching times are reduced by 50% after the first time a given query is executed.

7.2.4 Test 4 - Database performance and scalability

Goal: Determine how fast the most common database queries in Shredhub executes
Tests: Database speed
This test investigates the speed differences in using a document-oriented database versus a SQL database for Shredhub. The test is done by timing the query times spent when a User performs a given action that requires a database operation. A set of the most commonly used database queries are used as test cases. The timings start from the time the query instruction is made by the calling database handler, and ends when the result is mapped to a domain object (i.e a Java object for Architecture 1.0, and JSON object for Architecture 2.0). The test cases are designed to inspect at least one query from each of the four CRUD operations. Multiple Read operations have been checked, considering there are many types of read operations in Shredhub. The results are showed in table 7.2.4. The results are fairly equal, however, there are some differences. Operations for Architecture 2.0 that requires MongoDB to perform custom join operations, are generally slower. This concerns ShredsByFanees, and ShredsByFaneesOfFanees. Operations that require SQL to join while MongoDb does not have to join because of its nested structure results in performance gains for Architecture 2.0. This concerns ShredsByTags and create, update, and delete operations.

7.2.5 Test 5 - Code Flexibility Test

In this section we propose a test case that is meant to test the code flexibility and maintainability in the two prototypes. In this test, a new interactive user feature is to be implemented on Shredhub. The test is designed in a way that involves the modification of the User interface, implementing a new business process, and alteration of the database. The results for this test outlines the number of code statements that were added and modified, and the programming language(s) that were used to implement the feature. The new user feature that was implemented is given below.

User feature: The Guitar Showroom The Guitar showroom is part of the User's profile page, where the User can have pictures of his guitars, and other Users can view the images one-by-one by scrolling sideways using left and right arrow buttons. The scrolling must be highly interactive, meaning no page refresh can happen. Also, for every picture, the User can choose to "dig" the guitar. By clicking "digg", the guitar earns a "digg" point. Digg points is a way for the User to show that he likes the guitar. Each time a User diggs a guitar, the owner also earns one experience point. The User is not allowed to dig one of his own guitars.

For testing purposes, a set of the existing test Shredders are to add an image for one of their guitars, and a fictive digg-value (i.e change some of their guitars to showroom guitars).

Result for Architecture 1.0

The code that was added and modified in order to implement the feature is given in the table below:

Jsp + 47 lines (Html + Jsp script statements)

JavaScript + 81 lines.

Java + 73 lines, - 10 lines

SQL + 1 table, - 1 column + 1 column migration for every Shredder.guitars[]

In the JavaScript implementation, no code reuse was successfully implemented. Therefore, 4 new specialized and tightly coupled JavaScript functions were added to the front-end codebase. The back-end however, did not require much modification, except for the database mapping code that was required in order to handle the change in the SQL structure.

In the Java codebase, a new controller handler, service function and DAO function had to be implemented. In addition, the ShredderMapper class had to be modified in order to handle the update in the SQL structure. Also, a new domain class had to be created to hold the new Sql update: GuitarForShredder.java. Now, every Shredder has an array of GuitarForShredder objects, as opposed to earlier, when every Shredder had a simple String array of guitar names.

In SQL, a new table had to be generated to hold an image string, digs int, and name string, and a reference to the Shredder who owns the guitar. Previously every Shredder table had a simple array of strings that represented the guitars they own. Now, in order to implement the new update in Sql, I first had to create the new table GuitarForShredder, then for each Shredder, move every guitar column into a new row in the GuitarForShred table, then finally remove the guitars array in the Shredder table. The alteration of a Shredder is given in the example below:

```
1  // Old Shredder table:
2  Column   | Type   |
3  guitars | text[]   |
4
5  // Showroom guitar
6  Column   | Type          |
7  guitar   | character varying(50)    |
8  shredderid   | integer         |
9  imgpath     | character varying(20)   |
10 digs          | integer         |
```

Result for Architecture 2.0

The code that was added and modified in order to implement the feature is given in the table below:

Html + 40 lines - 4 lines

JavaScript App + 88 lines.

JavaScript API + 19 lines

A lot of view logic code on the front end was successfully reused. A couple of new event-handlers were added to the Shredder view, and a business operation was added to the Shredder model.

On the back-end, there was no need to alter the database, because of mongoDB's flexibility: Previously, every Shredder had an array of guitars as strings that represents their list of guitar (only the names, just like Architecture 1.0). Now, for every guitar that is to include an image and a dig int, the array index that used to represent that particular guitar could simply be altered to be a nested JSON object inside the array, instead of a simple string. Now the only necessary modification in MongoDb was the alteration of the guitars Array for those test Shredders that was to change a guitar from being just a name to a JSON object with name, image and dig value. There was no need to alter any of the other Shredders, or even alter any of the CRUD operations that touches the Shredder object. The alteration of a Shredder can be seen in the code below:

```
1  // Old guitar array:
2   Shredder {
3    guitars : [''Gibson Les paul'', ''Fender Stratocaster'']
4  }
5
6  // Showroom guitar
```

```
7  Shredder {
8    guitars: [''Gibson Les Paul'',
9    { name : ''Fender Stratocaster'',
10   image : ''fenderStrat.jpg'',
11   diggs: 34
12   }]
13 }
```

Also, there was no need to alter the Shredders that had not yet added switched to showroom guitars, because the program now accepts both guitars as simple strings, and JSON objects. The only modification needed at all was actually in the Html template that uses the list of guitars: an if-else block has to be added in order to check if a guitar is a string or a JSON object. Hence the -4 Html lines in the listing above.

Also, the back-end had to add a new REST api function to handle the database update for adding a dig.

7.3 Summary

In this chapter we have looked at the tests that were made in order to analyze and compare the two prototypes. Five different tests were created which analyzes performance, scalability and code flexibility/maintainability. The results mostly proved advantages for Architecture 2.0, but Architecture 1.0 also had some winning features. The results will be discussed in the following chapter.

Part III

Discussion and Conclusion

Chapter 8

Discussion

In this thesis I found that rendering Html on the client takes a lot load off the server, and therefore makes it more scalable, because less processing has to be done for each request. This mostly resulted in better response times because the client could choose only to render the parts of the page that are necessary, and doesn't have to ask the server for the Html. I also found that Web-apps can perform better by having state and business logic in the client, because it endorses the storages of database objects in the browser's memory, and therefore reduces the amount of server calls needed. Once it must consult the server, it happens asynchronously in the background without the user noticing any delay. It also has scaling benefits because the server doesn't have to maintain session data in memory. The programming benefits for this is that both view logic and business logic is implemented in the same language, which creates a more coherent code base. Finally I found that the SQL implementation performs faster for read operations that doesn't use joins. In those cases, MongoDB was faster because much data was gathered in the same collection, which avoids having to join multiple collections in queries. Also, using Redis was a good solution for the stateless server model, because it authenticates each request very quickly. Both these databases facilitate a very satisfactory programming environment on the back-end, because only one programming language is used.

8.1 Page Rendering

Test 1 and 2 clearly shows that rendering on the client gave faster response times for the User. The reason is that the browser maintains all the Html that is needed for the whole App. Therefore it doesn't have to ask the server for Html when the User goes to a new page or performs an actions that requires new Html to be rendered. It can just be fetched from the browser memory, and merged together with the necessary JSON data. The client still has to fetch the JSON data from the server from time to time, before it can do all the rendering, but this is much less data then a complete Html page. Good examples are figure 7.3 on page 128 and 7.4 on page 129, where the page is rendered as soon as the browser receives JSON data from the

server, and the App has modified the Dom to display the new result. Also, when the client performs the rendering, it can choose to render only the parts that are necessary in order to display the result of a user action. In the case for server side rendering, it has to render the whole page in any case. In addition, this leads to browser page refreshes, which is a unfortunate user experience.

Rendering on the client also limits the amount of bandwidth consumed, which is proved in Test 1 in the previous chapter. Looking at the numbers regarding data received, shows that JSON data representing only the data needed to display a particular request (Architecture 2.0) is generally much less of a data quantity then the complete Html page rendered (Architecture 1.0). An important decision is wether to eagerly fetch all the Html at first, or lazy fetch Html when its needed. I argued that eager fetching was preferable for Shredhub, because when all the Html templates were merged together and minified, the complete size was small enough to send in the initial page load without harming this request too much. However, if Shredhub is to grow extensively in size with new pages and user features, it will probably be desirable to fetch Html lazily. A disadvantage with server-side rendering is that it is a time consuming process, something that is clear from figures 7.3 on page 128 and 7.4 on page 129 where a lot of time is spent on the server. An outcome of this is that Architecture 2.0 can handle many more simultaneous users, because the average request times are much higher (table 7.2.3 on page 133 and 7.2.3 on page 133), and more scalable.

Now, figure 7.1 on page 127 shows that rendering on the server results in quicker response time for the initial page request. The reason for this is that in Architecture 2.0, the browser has to wait for the whole app to be completely loaded in the browser before it can start fetch additional JSON data and render the page. This is very time consuming and the tradeoff can in some cases be too high, depending on where one sets the upper limit for initial page response times.

8.2 State and Business Logic on Client

Moving application state and business logic to the client has clear performance advantages for Shredhub. The browser stores state data in its JavaScript memory and local storage, which in many cases avoids the need to consume the server for data. Good examples that proves this are in figures 7.5 on page 130 and 7.5 on page 130, in where the browser doesn't even have to fetch data from the server. When the client does have to fetch data, it consumes the API, which is always done asynchronously with Ajax, and in effect doesn't lead to any browser page refreshes. The outcome is a highly interactive user experience where the user doesn't notice the http requests.

One disadvantage with this, however, is that it requires a lot of JavaScript code in order to keep the code base maintainable, which might lead to a somewhat large amount of JavaScript statement to execute in

order to perform a simple task. Figures 7.6 on page 131 and 7.7 on page 131 showed the advantages of the simple JavaScript handlers in Architecture 1.0, which led to slightly better response times. However, the tradeoff is that these simple JavaScript handlers does not facilitate code reuse or any modularity, because they are implemented as simple handler functions that merely does one specific task. In the long term, this might add up to tangled and messy code. A clear indication of this was found in test 5 in the previous chapter, where 81 lines of JavaScript code was separated in 4 and tightly coupled event handler functions was implemented, in order to build a new interactive user feature for Shredhub. Architecture 2.0, on the other hand has a more intuitive and coherent code base, because both the view logic and business logic is implemented in the same language, and cooperates neatly with the MV* design pattern. Architecture 1.0 uses two different languages for view and busines logic (Jsp, and Java), and much of the view logic is tightly coupled with the Html.

Another advantage with this architecture is that the use of the back-end API decouples the client from the server. In Architecture 1.0, each Html form and url anchor tag has an associated controller handler on the server. The API however, is more general in that it doesn't return a view, because it is up to the client caller to decide how to use the results. Also, most API calls are flexible in that they allow the caller to for example define result sizes and page numbers. Now, the big advantage with this is that it allows other client users to use the API as well. This could be a future mobile app, or a 3rd party application that wishes to use Shredhub data.

Now in addition, avoiding sessions on the server had a very high performance impact. In Architecture 1.0, the server had to maintain a large set of Java objects in memory, for every current user. This resulted in slow response times when the number of simultaneous users are high, and the server was not able to handle more then 600 simultaneous active users (see figure 7.2.3 on page 133). Now, it is difficult to state how much of this limitation was caused by the amount of memory consumed for maintaining state and sessions on the server, and how much was due to the complex rendering processes that happens for every request. I acknowledge the fact that this thesis lacks a deeper inspection of this in order to be able to draw more concrete conclusions regarding the scalability issue for Architecture 1.0. A solution would be to use a profiling tool to inspect how much time was spent on state handling, versus template rendering, and to use a monitoring tool to verify the amount of memory consumption used in maintaining session objects.

A disadvantage with having the server completely stateless becomes clear in figure 7.2 on page 128, in which case a lot of time is spent on the server fetching JSON from the database. Now, this is done over 8 different http requests, which results in a much transmission overhead and slow response time. Even though the complete rendering process is faster for Architecture 2.0 from the User's perspective, a lot of time could have been saved if the server was aware of the 8 different database fetches that are required to display the shredpool. Another solution would be to offer a more coarse-grained API function that simply fetches all the JSON data that

is needed in order to build the Shredpool page on the client, given a Uid for the Shredder. The tradeoff here is that offering such coarse-grained API functions could create tighter couplings between the client and the API, similar to Architecture 1.0.

Although the study shows many advantages for this thick client architecture, there are some major pitfalls:

- Some business rules has to be duplicated on the server in order to prevent malformed user input. This could come intentionally from users who knows how to issue Http requests without using Shredhub.com's web interface. Now, this was only implemented for a few Http requests in Architecture 2.0; just enough to make me aware of the drawback.

- The Web app might not perform as well on other client machines and browsers then the ones that was used for testing. This might be a serious pitfall, because the result might be that slow computers and/or old browsers executes the JavaScript code so slow that Architecture 1.0 might be a preferable solution in terms of performance. This is most likely a case for old smart phones and desktop computers. I do also acknowledge the fact that the testing phase of this thesis should have been done more extensively, on various computers and smart phones in order to support these accusations.

- The architecture is also not optimal for search crawlers, in where crawlers inspecting Shredhub would find merely empty Html tags without content. Now, this is not a very critical problem, because most of Shredhub's content is only to be viewable once logged in. However, some parts of Shredhub should be fully searchable on the web, and hence a better solution for this problem remains to be solved.

8.3 NoSql vs Sql

The database test for Shredhub was somewhat limited, however the results show some valuable points that are worth discussing. First of all, the MongoDB implementation does not perform particularly fast when manual join operations have to be done. This concerned the two read operations get Shreds by fanees, in which the JavaScript caller first has to fetch the Shredder, extract his array of fanees, then do a fetch operation for all Shreds where the owner is in the set of fanees. An even worse case is when it has to fetch Shreds made by fanees of a Shredder's fanees, I.e a second degree graph search. Here, the JavaScript caller has to further fetch all the Shredder's fanees' fanees and search for Shreds where one of them is the owner. Now, I acknowledge the fact that I should have added an index for the owner for a Shred in order to speed up this execution. Now, in PostgreSQL, the Shred owner is a foreign key, in which the dbms has

already added an index for it. However, even higher speed results could have been achieved if I had chosen to add indexes for quicker sorting; for example, I should have added an index for the time a shred was created, considering this is used as sorting key for most Shred and Shredder queries. This applies to both the SQL and MongoDB implementation. Another option for the MongoDB implementation was to further de-normalize the documents and investigate the possibilities to even further duplicate code in order to avoid doing manual joins. I acknowledge that I should have spent more time in the beginning, on designing a more performance-oriented document model.

Another advantage with the SQL implementation is that it successfully applies caching techniques for certain read queries. Therefore, subsequent Shred-read queries after the initial ones, were cut down with at least 50 %. Now, it maybe that this is only a matter of configuring the MongoDB server, however, I have not spent further time investigating this.

On the other hand, MongoDB has clear speed advantages in cases where join operations are avoided. This happens because the MongoDB implementation wraps many of the separated SQL tables from Architecture 1.0 into one big object, and therefore avoids having to join multiple tables together. The results shows that create, update and delete operations are generally faster on Shredhub, because they are all just operations on a single MongoDB document, as opposed to the separated tables in SQL. Even the range query "get Shred by tags" are faster, because tags is a nested string array inside every Shred. In Architecture 1.0, tags is a separate table in which case joins has to be done.

Also, Architecture 2.0 has a big programming satisfactory advantage. Everything is written in JavaScript. This facilitates better and cleaner cooperation across the whole codebase. This is especially beneficial for the database wrapper, because the data structure used is JSON based, which makes possible to completely avoid database mappers. In Architecture 1.0, database mapper code makes up the majority of the back-end codebase. Now, because JSON is the data structure used as transmission medium in the API as well, no marshaling is needed here either. Hence, the Architecture saves a lot of source code lines. The point is proven in the number of lines in Architecture 2.0 versus Architecture 1.0: 10505 vs 15867 lines.

A final observation is that using Redis made it possible to have a session free server, and still authenticate users for every API request (except requests for the login page), without getting performance bottlenecks. A great advantage with this is that it facilitates a shared-nothing architecture, which again facilitates distributing the back-end to many server machines. This could lead to large performance and scalability gains. This is somewhat limited in the session-oriented architecture, because sessions does not apply so well in distributed deployments: If a session is created on one server, and the user is directed to another server in a later request, the session is not found, and the user will be directed to the login page. Also, if a server goes down, every sessions on that machine is lost. Now, there are solutions for these problems, but they still provide more distribution

obstacles then Architecture 2.0 does.

8.4 Strengths and Limitations of the Study

8.4.1 Srengths

The thesis reveals many important aspects of modern Web architecture design. I believe the results are not specific to Shredhub, but demonstrates principles that are valuable for traditional Web-apps. The thesis especially proves one important point: Moving demanding concerns to the client using JavaScript is not only feasible, but leads to increased scalability and higher back-end throughput. This trend is still very newfangled, and lacks research. I also believe that the thesis reveals pros and cons for both architectures, and is not biased by my own experience.

8.4.2 Limitations

One limitation with this study is that I do try to solve many different problems in one single project. One could argue that each problem statement does not have sufficient material to give significant conclusions. I mean that the results from the tests are evident enough to answer the problem statement, however, I do acknowledge that I could have chosen to narrow the scope of the thesis. Parts of the reason why I studied all these different technologies was that I found it very educational. I should have tried to team up with another master's thesis student who could have done the back-end or front-end part of the study.

Also, the code flexibility/maintainability test is a bit limited. This could even be a complete thesis in itself. The results do prove some important points however; the importance of structuring JavaScript code on the front-end in order to avoid ending up with tangled and non-reusable JavaScript functions. Something that is fully possible with the JavaScript language itself, or by using open source frameworks.

8.5 Implications for Practice

I believe this study is important, because there are still many Web application developers who swear to the concepts of *reference-model 1.0*. Also, many developers aren't aware of the possibilities of the JavaScript programming language itself, for example that it is fully possible to build modular and flexible codebases. The thesis advocates the capabilities of modern browsers, which enables developers to build thick client JavaScript Web apps. These thoughts are fairly new, and lacks research.

Even though the thick client architecture doesn't necessarily fit every modern Web application, they are important concepts to contemplate. For example is rendering Html on the client very relieving for the server, and leads to increased scalability and throughput. If a complete client-side rendering architecture is not an option, maybe choosing to perform some

client-side rendering and some server-side rendering is possible. I leave this question as a task for further research: The study of finding solutions for combining server-side rendering with client-side rendering. This could also help improving the results picked up by search crawlers, something that is very important for applications that are not behind a login barrier.

In this thesis we saw pros and cons for both using MongoDB and SQL. Maybe, there are ways to combine these in the same Web application, with the purpose for finding hybrid solutions. We saw for example that MongoDB could be used in combination with Redis in Architecture 2.0. This is an open question, and I leave with an encouragement to further look for application areas where noSQL solutions can be combined with SQL.

Finally, we saw that Architecture 2.0 could be built by using JavaScript both on the back-end and front-end. A problem, however, was that some business rules had to be implemented both on the back-end and the front-end. I did not try to look for a solution to merge these together with the intentions for avoiding code duplication. I do believe this could be possible considering there is one overall programming language, so therefore I leave this as a further research topic.

Chapter 9

Conclusion

In this thesis we have investigated traditional and innovative architectural principles for modern Web applications. We defined a Web app as a traditional Web 2.0 application that includes highly interactive behavior, social networking features, and big quantities of persisted data. The problem statement concerns architectural principles for implementing such applications. It asks the question of wether the application can benefit for doing Html rendering on the client, and if there are advantages for moving business logic and state handling to the client. The motivation for this is the lately improvement in JavaScript engines in modern browsers, which enables more complex JavaScript executions on the client. Also, the problem statement addresses modern database solutions called noSQL, and asks wether there are any such database that suits the JavaScript-oriented Web application.

In order to solve this, we defined two reference-models; *Reference-model 1.0* and *Reference-model 2.0*. These addresses principles for respectively a traditional Web application architecture, and a modern and innovative Web application architecture. *Reference-model 1.0* states that all application processing happens on the server, including state handling, business logic and page rendering. In addition, the data is persisted with a relational database. *Reference-model 2.0* states that these concerns are now completely implemented on the client, using JavaScript. In addition, the data is persisted by using noSQL database technologies.

The principles for *Reference-model 1.0* and *Reference-model 2.0* were applied in the implementation of two different architectures that solves the same problem domain; a Web app called Shredhub. These two architectures were respectively called *Architecture 1.0* and *Architecture 2.0*. In order to solve the problem statement, a set of five extensive test case were designed and performed on these architectures.

The results showed that rendering Html on the client is fully feasible, by fetching all the necessary Html to the browser on the initial Web-app initial. Client-side JavaScript is implemented to choose to render only the Html that is needed for every subsequent request, thus avoiding to consult the server for Html. This lead to a much more scalable back-end where requests where handled very fast, because the amount of work done on

the server is very little. A problem however, was that the initial page load was very slow, because a lot of JavaScript data had to be fetched from the server. Also, one obvious programmer satisfactory aspect came clear for Architecture 2.0. It was easier to separate the view-logic out of the Html templates, because the view-logic is implemented in the same language as the business-logic. Therefore, the view-logic and business logic concerning the same domain, could be implemented under the same module, and thus cooperate better.

The tests also showed that moving state and business logic had big performance and scalability gains. The front-end could avoid consulting the server, and when it did, the server merely had to query the database and send small-sized JSON data back. The results were that the front-end had very quick response times for user actions, and much work was released from the server. The latter led to higher scalability, where many more concurrent users could be served then in Architecture 1.0. A pitfall here is that Architecture 2.0 does require an efficient browser and client machine in order to run efficiently. Another disadvantage is that some business rules has to be implemented both on the client and the server. However, we proposed a hybrid solution for a future study, where the code base can be shared on both the client and server, and thus avoiding the need to duplicate.

As for the databases, both SQL and noSQL had their advantages. SQL were slow in cases were many join operations were needed. MongoDB could be implemented in a way that avoids the need to join, by having fat objects that contains all the necessary data most query requests. However, once multiple objects had to be joined together, MongoDB could be significantly slow. Another advantage with Architecture 2.0, was that MongoDB handling could be implemented without the need to have data mappers. This is a big programmer satisfactory advantage, because it reduces the size of the code base, and simplifies working with the database. We also saw that Redis could be used to support the stateless server, by quickly authenticating every Http request. Architecture 1.0 uses server-side sessions to maintain authentication through an Http session, something that can lead to slow response times because the sessions consumes a lot of memory. Avoiding this with Redis had great performance results for Architecture 2.0.

Bibliography

[1] URL: http://www.w3.org/History.html.

[2] URL: http://www.totic.org/nscp/demodoc/demo.html.

[3] URL: http://www.w3.org/Style/CSS/.

[4] URL: http://javascript.about.com/od/reference/a/history.htm.

[5] URL: http://msdn.microsoft.com/en-us/library/hbxc2t98(v=vs.85).aspx.

[6] URL: http://support.microsoft.com/kb/154544.

[7] URL: https://www.google.com/intl/us/about/.

[8] URL: http://www.mozilla.org/en-US/.

[9] URL: http://www.webpronews.com/javascript-leads-the-pack-as-most-popular-programming-language-2012-09.

[10] URL: http://www.facebook.com.

[11] URL: http://www.Twitter.com.

[12] URL: http://www.pinterest.com.

[13] URL: http://www.airbnb.com/.

[14] URL: http://www.amazon.com/.

[15] URL: http://www.coursera.org.

[16] URL: http://mike2.openmethodology.org/wiki/Big_Data_Definition.

[17] URL: http://nosql-database.org/.

[18] URL: https://developers.google.com/maps/.

[19] URL: http://httpd.apache.org/.

[20] URL: http://www.iis.net/.

[21] URL: http://rubyonrails.org/.

[22] URL: https://www.djangoproject.com/.

[23] URL: http://www.w3.org/XML/.

[24] URL: http://www.webkit.org/.

[25] URL: https://developer.mozilla.org/en-US/docs/Mozilla/Gecko.

[26] URL: http://www.oreillynet.com/pub/a/javascript/2001/04/06/js_history.html.

[27] URL: http://www.crockford.com/javascript/javascript.html.

[28] URL: http://www.microsoft.com/silverlight/.

[29] URL: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications.

[30] URL: http://www.jquery.com/.

[31] URL: http://www.prototypejs.org/.

[32] URL: https://code.google.com/p/v8/.

[33] URL: https://mail.google.com/.

[34] URL: https://maps.google.com/.

[35] URL: http://www.websocket.org/.

[36] URL: http://www.json.org/.

[37] URL: http://www.w3.org/XML/.

[38] URL: http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html.

[39] URL: http://www.microsoft.com/en-us/sqlserver/default.aspx.

[40] URL: http://www.postgresql.org/.

[41] URL: http://www.redis.io/.

[42] URL: http://couchdb.apache.org/.

[43] URL: http://www.mongodb.org/.

[44] URL: http://cassandra.apache.org/.

[45] 1997. URL: http://www.windowsitpro.com/article/news2/microsoft-and-spyglass-kiss-and-make-up-16683.

[46] 2008. URL: http://info.cern.ch/.

[47] Daniel J Abadi. 'Data Management in the Cloud: Limitations and Opportunities'. In: *IEEE* (2009).

[48] David Thomas Andrew Hunt. *The Pragmatic Programmer: From Journeyman to Master*. Addison Wesley, 1999.

[49] David Patterson Armando Fow. *Engineering Long-Lasting Sotfware: An Agile Approach Using SaaS and Cloud Computing*. Strawberry Canyon LLC, 2012.

[50] Severance C. 'JavaScript: Designing a Language in 10 Days'. In: *IEEE Computer Society, (Vol. 45, No. 2) pp. 7-8* (2012).

[51] et al Chang Fay. 'Bigtable: A distributed storage system for structured data'. In: *ACM Transactions on Computer Systems 26.2: 4* (2008).

[52] E. F Codd. *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[53] P Crane D McCarthy. 'Comet and Reverse Ajax: The Next-Generation Ajax 2.0'. In: *Apress* (2008).

[54] Thomas E. 'Service-oriented architecture'. In: *Prentice Hall* (2004).

[55] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[56] Mikowski M and Powell J. *Single Page Web Applications*. Manning Publications Co, 2013.

[57] Johnson R. *Expert One on One J2ee Design and Development*. Wrox, 2003.

[58] a division of VMware SpringSource. *Spring documentation, Chapter 6, Validation, Data Binding, and Type Conversion*.

[59] a division of VMware SpringSource. *Spring security framework*.

[60] O'Reilly T. 'What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software'. In: *Communications & Strategies, No1, p17, First Quarter 2007* (2007).

[61] Wetherall D Tanumbaum A. *Computer Networks*. Pearson Education Inc., 2011.

[62] Jeremy Viegas William G.J. Halfond and Alessandro Orso. 'A Classification of SQL Injection Attacks and Countermeasures'. In: (2006).