UNIVERSITY OF OSLO
Department of Informatics

Developing Modern Web Applications

A Comparison of Traditional and Modern Approaches

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 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 webapp. 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.

1.2 Goals

Lately, there has been a dramatical change in the types of applications that are hosted on the Internet. These applications offer highly interactive behavior and supports huge amounts of simultaneous users. At the same time, software developers tend to rethink the way traditional web applications are designed, and a lot of new architectural proposals and web technologies has emerged with this new web paradigm. The goal of this thesis is to explore such an innovative web architecture and investigate the pros and cons of a modern web solution versus a traditional web architecture. It so happens that traditional thick server architectures are still commonly seen on the web, and so it is relevant to identify any advantages these architectures might have over a modern thick-client solution.

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.

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 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.

1.6 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.

1.7 Contributions

1.8 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.

1.9   Criteria

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
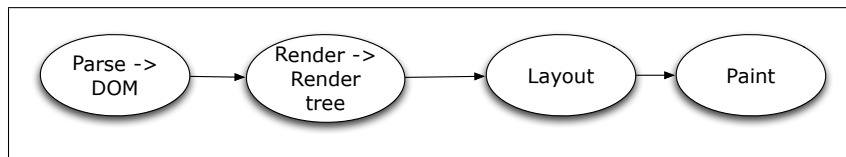
Figure 2.1: The rendering engine's responsibility

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.

23

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   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 5.2 on page 79. 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.

2.3.6 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.

2.3.7 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.

2.3.8 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, 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.

2.3.9 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.

31

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 schemaless, 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.

Document-oriented databases
    Is a datastore that is based on documents that contains unstructured

content. 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 overall architecture, both seen from a hardware and software perspective.

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 sends a 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 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

---

[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.

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. Often the URL request 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 input 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.

TODO: Discuss front-controller vs MVC vs other shiet

The domain layer is responsible for executing the business functions that are supported in the web application. These are the functions that makes up the core services of the application. The domain 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. The business operations in the domain layer uses the domain object, by performing various actions 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 a domain layer. For example in the domain model design pattern [55], the domain objects knows how to perform business operations on themselves. While in the transaction script pattern [55], the business logic is encapsulated in separate functions that operates on the domain objects. The business operations (or transaction scripts) are independent procedures that has a one-to-one mapping with the actions supported in the presentation layer (e.g the web site). The domain objects are completely stripped from any business logic functions, they merely contain the data attributes as field values. The domain objects are usually encapsulated in the first-order citizen types used by application's programming language. For instance in Java or Ruby, a book would be represented in its own Book class, a movie would have a Movie class, an order would have an Order class etc.

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

---

[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.

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 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. 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 acts as thin classes that simply delegates business logic to the Models. The model classes are built around the domain model design pattern, meaning they will perform business operations directly on themselves. 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 Ruby model objects, 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
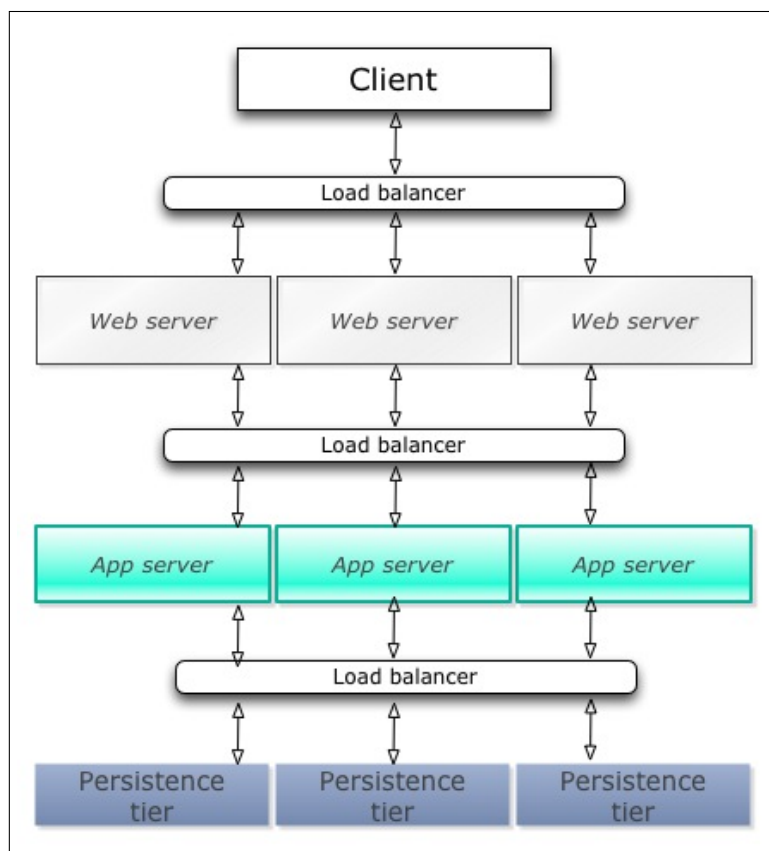
Figure 3.1: A distributed production architecture

support, and the framework will in return 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 performs CRUD operations on the appropriate 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 from a Model class. 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 is built with the active record pattern, so that every Model knows how to perform database operations on itself. 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   Just like with Rails, SpringMVC is also built around the MVC pattern, however the controller part is built using the **Front controller** pattern. This pattern works by having a single entrance point for all the HTTP requests, that sits and consolidates actions to various routines that is to be executed for each URL. This can be seen in figure 3.2 on the facing page.

When a request comes in, the front controller will ask a **handler mapper** to get a reference to a specific **controller handler** based on the URL. In addition the handler mapper performs a set of pre- and post processing procedures, like validating the input from an HTML form, or marshaling an XML object to a Java class. The front controller then receives a reference to the controller handler, and the front controller will further delegate the request to this handler. The controller handler 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**
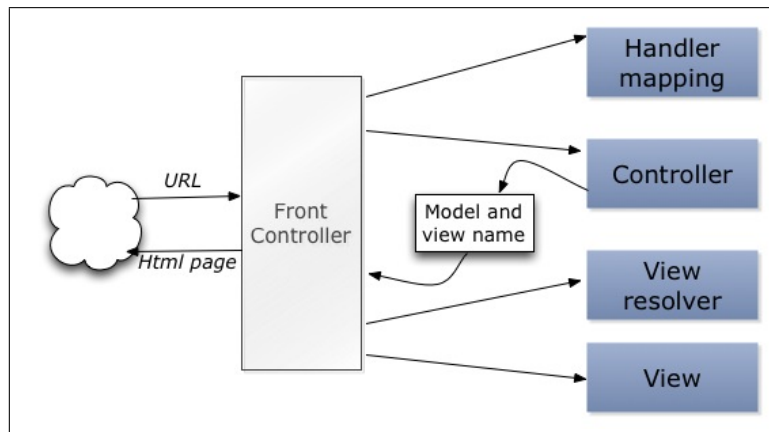
Figure 3.2: A request flow with spring MVC

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. This could be by lazy loading (the object isn't instantiated until accessed), a singleton (only one instance of a class is allowed to be created), one instance for each session, or one for each HTTP request. The IOC container provides a component based structure for the application by following the dependency inversion principle [**dip**]. This principle states that object 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] This conforms to the convention over configuration principle, which leads to simple and quick web application development.

A typical domain layer in a spring application is built with the service layer pattern [**serviceLayer**]. In the service layer pattern, the business operations are 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. There are also **domain objects** that represents the main data entities in the application. These domain object are simple java classes with no logical functions, only private data attributes with accessor methods. The service classes uses the domain objects, and 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.

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 interation 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

---

[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.

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.

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 gradually being 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.

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 only time the server is contacted is when 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. The front-end tier has taken over most of the responsibilities that used to be done on the server, which now involves:

- 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 (ref to airbnb), 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

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].

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).

- 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.

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 work well in a cloud environment. The so-called noSQL paradigm is a common term for classification of database technologies that does not belong to the family of relational databases. These technologies has gained massive popularity

---

[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)

in the last couple of years, because is has showed to deliver excellent performance and scalability in cloud environments. In addition, many of the noSQL technologies are built to fit nicely together with JavaScript, which makes it easier for JavaScript based applications to manage the database.

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

3.3.1 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.

3.3.2 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.

3.3.3 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.

3.3.4 Modular JavaScript

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.

3.3.5 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.

3.3.6 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 by all browsers, except Internet Explorer 7 and earlier versions. HTML 5 web storage contains two storage containers: local storage and session storage. 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 session storage up to 10 mega bytes, while a cookie is limited to 4 kilo bytes. The session storage 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.

3.4 Our 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 from these reference models and applied them in two different software architectures for a prototypical web app. Our first solution is called Architecture 1.0, while the latter is called Architecture 2.0. The first one follows a three-layered architecture that is deployed on the backend. 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 thin client architecture that is built with a sibling of the Model-View-Controller pattern, which runs entirely on the client. The backend is just a simple repository for manipulating the database, which is a noSQL database. Just like in Architecture 1.0, the backend here is also built with Spring

MVC. However, the code that runs on the backend of Architecture 2.0 is something way (less, and) different from what's deployed on Architecture 1.0.

3.5   Summary

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 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, 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

Chapter 4

Shredhub, a Web 2.0 Application

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 web app has been built twice from the ground up with two completely different architectures. The first one conforms to a traditional approach, and the second conforms to a modern approach. In this 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.

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 user-credentials. 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.

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, where the Shredder can watch a set of recommended shreds. The set of Shreds that are displayed for the Shredder is chosen based on the the Shredder's fanees, the types of Shreds he likes, and the current most popular Shreds on the site (which is equal for all the Shredders in the app). There is also a section with the latest *"shred-newsitems"* that are customized for the Shredder. These are categorized into four: newly created Shreds made by a fanee, new Shreds made in a battle (also made by a fanee), a fanee recently entered a new battle, and a new Shredder that might be of interest to the given Shredder, just started using Shredhub. The latter is meant as an encouragement for the Shredder to widen his fan graph. 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 4.2 on page 63. If the Shredder clicks on a Shred, a new

---

[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
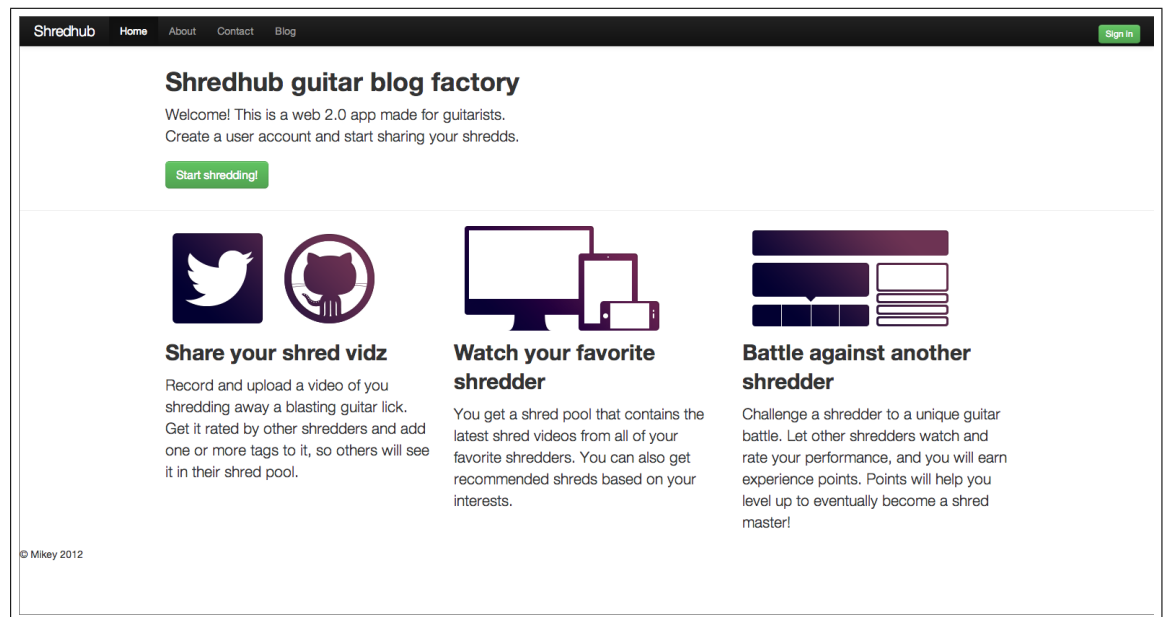
Figure 4.1: The front page at Shredhub

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. 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. The way experience points is calculated is based in a simple mathematical function that takes into account the number of Shreds the Shredder has uploaded, how many fans the Shredder has, and the ratings the Shredder has received for his Shreds.

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 4.3 on page 64.

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

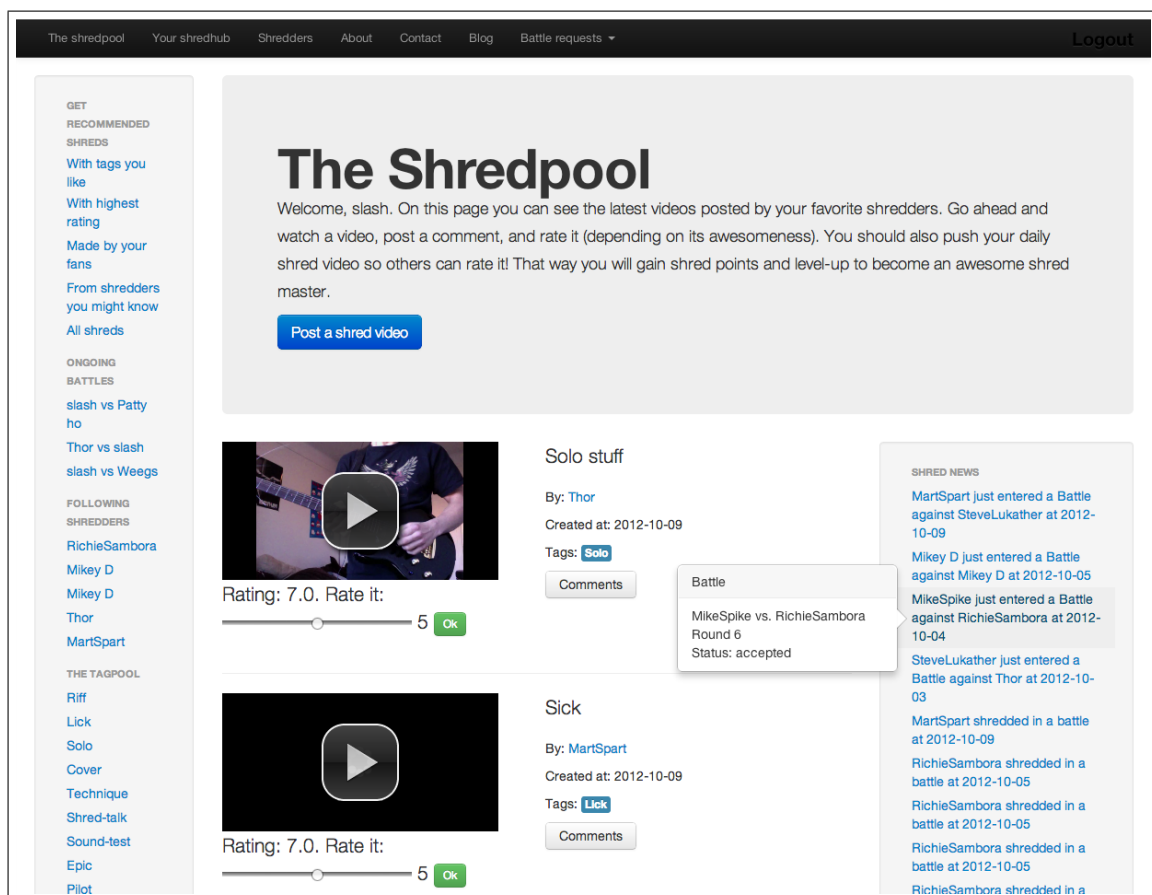This is a page that displays the details for a given Shredder, that has
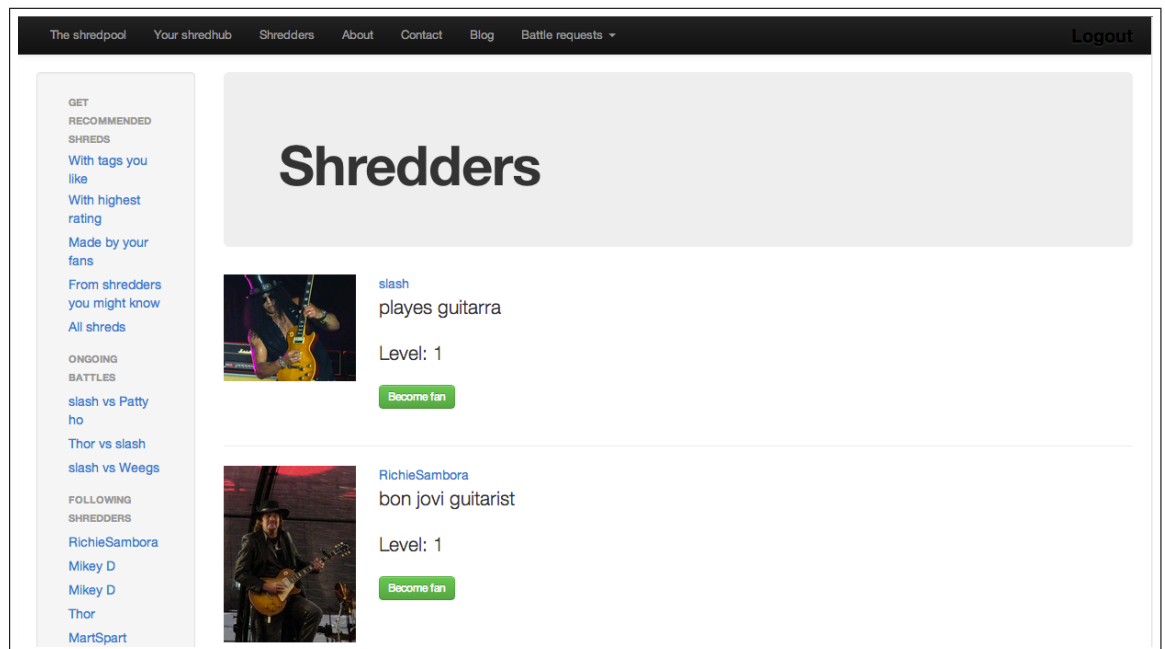
Figure 4.2: The main page in Shredhub

Figure 4.3: The list of shredders

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 (4.4 on the facing page).

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. This is displayed in figure 4.5 on page 66. In a battle I distinguish between the battler who initiates the battle, and the battlee, being the one who is challenged

Figure 4.4: A shredder's profile page

Figure 4.5: Displaying a battle between two shredders

Chapter 5

Architecture 1.0

5.1 introduction

In the previous chapter we looked at the user requirements and behavior for Shredhub, our web 2.0 application. In this chapter, and the next one, we will have a detailed look at the software architectures and technologies that has been chosen implement that application. The application has been built twice, with two completely different architectural approaches. The architecture outlined in this first chapter, is named Architecture 1.0, and it is designed with a traditional approach in mind, following the principles from *reference-model 1.0*. The next chapter discusses the architectural details in Architecture 2.0, which is based on *reference-model 2.0*.

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 the application server. The app is divided

Figure **??** on page ?? shows my first architecture attempt. This is a classical monolithic three-layered architecture. In this scenario, Spring MVC is to be found in the presentation tier. This is where the controller functions resides. When a url request comes in, the framework will delegate the call to the appropriate controller function, as described in chapter 2.

The controllers validates data and further delegates calls down to the middle tier. This tier is responsible for handling business logic. There are many variations of how to design a middle layer, which will be discussed later in this section. The middle tier commands the data tier to perform CRUD operations. These operations must be atomic, so the middle tier would need to do transactional management to make sure database operations are either totally completed or not at all. The data tier is responsible for mapping java objects to SQL strings that are written and read from the database.

5.3 The presentation tier

The presentation tier is the part of the application that receives a url and renders a particular view back to the client. This module is usually referred to as the controller part in the MVC design pattern. The controller function's responsibility should be simple and cohesive. Therefore, its
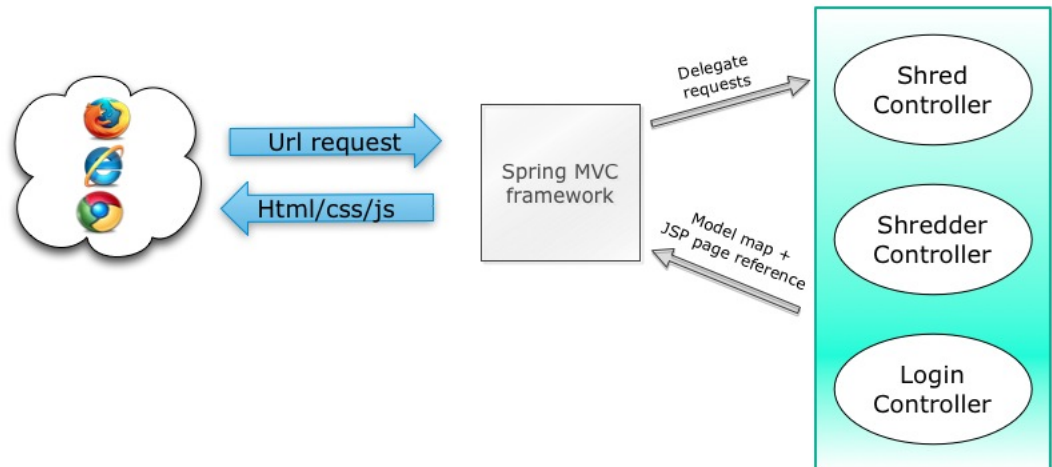
Figure 5.1: The presentation tier's responsibility

only responsibility is web specific only, including input validation, and delegation of requests to business functions residing in the model layer (or middle tier in figure ?? on page ??). When the business process returns, the controller prepares a map of data objects that the Spring framework uses to populate an html page (this map is called the *Model* object). The model object can store all kinds of objects (in my case Shred, Shredder, Battle etc), but the jsp pages can only reference attributes in the model that can be accessed through a getter method. In other words, the jsp pages cannot reference any functions on the server. Finally the controller will return a name for the jsp page that the Spring framework will use to render to an html page that is to be sent back to the user. This is showed in figure 5.1.

There is exactly one controller handler function for every action (or url) that the application supports. Controller handlers are identified by a unique url mapping, as showed in the code below:

```
1   @Controller
2   @RequestMapping("/shred")
3   public class ShredController {
4
5       @Autowired
6       private ShredService shredService;
7
8       @RequestMapping(value="/{shredId}", method = RequestMethod.
            POST,
9       params="action=rate")
10      public String rateShred(@PathVariable int shredId,
11                  @RequestParam("rating") int rating,
```

```
12              Model  model ){
13      try {
14        shredService.rateShred(shredId,  rating );
15
16      } catch (IllegalShredArgumentException e) {
17        model.addAttribute("errorMsg", "Failed  to  rate  shred:  "
18         + e.getMessage()  );
19        return "errorPage";
20      }
21      return "redirect:/shredpool";
22    } }
```

The shredService reference is the gateway to the middle tier. Spring
will inject a concrete implementation of ShredService through its IOC
container (the @Autowired field indicates that an implementation must
be injected here). Here the user has requested an http post re-
quest to rate a shred with a given Id. The url could look like this:
http://www.shredhub.com/shred/1234/?action=rate&rating=6. When the re-
quest comes in, the controller function would simply delegate the call
down to the service layer. Upon return, the controller would potentially
catch an exception if something went wrong down the line. For instance
if the shredId does not exist, the controller will catch the exception, create
a proper error message, and return an error page back to the client. The
view resolver (as explained in chapter 2) will see that the response is a re-
direct, in which instead of rendering the jsp page that is mapped to the
name shredpool, it will ask the client browser to redirect to this url, which
eventually would end up being a html page that is sent back to the browser.
It is common to send redirects back to the client in http POST requests, in
case the client would try to refresh the page before the request has finished.
This would trigger a duplicate event on the server, which could lead to un-
wanted behavior considering POST requests are usually idempotent [49].

Form validation Protecting html forms from malformed user input is
an important aspect of web applications. It helps the user to ensure he
has posted the data he wanted (humans often makes mistakes). Also, it
protects against malformed data that might have bad intentions. Even
though the jsp pages might have validation logic built into the html forms
browser-side, there are ways to manipulate the data that is sent with an http
request. 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. The approach I have taken to guard against such
attacks is by applying positive filtering, meaning I specify what's allowed,
and forbid everything else. Another approach is to do negative filtering by
actively looking for suspicious input patterns, and return an error to the
user if a potential attack is found. However the latter approach is not as
secure because it is hard to imagine all possible attack-forms [62]. Besides,
new forms of attacks might be invented in the future. A good example of
how I might do form validation is showed in the code below. This is the
handler that creates a new Shredder (*www.shredhub.com/newShredder*). The
controller handler shows the cleanliness of verifying that the input received

by the user is correct, using Spring best practices.

```
1    @RequestMapping( value ="/newShredder", method = RequestMethod.
        POST)
2     public String newShredder(
3         @Valid Shredder shredder,
4         BindingResult result,
5         HttpSession session,
6         @RequestParam(value = "profileImage", required = false)
            MultipartFile profileImage,
7         Model model) {
8
9       if (result.hasErrors()) {
10        model.addAttribute("errorMsg", "Error: " + result.
            getFieldError().getDefaultMessage() );
11        return "/errorPage";
12      }
13
14      try {
15        shredderService.addShredder(shredder, profileImage);
16      } catch (ImageUploadException e) {
17        result.reject("Error creating user: " + e.getMessage());
18        model.addAttribute("errorMsg", "Failed to upload the
            profile image " );
19        return "/errorPage";
20      }
21
22      return "redirect:/login";
23    }
24  public class Shredder implements Serializable{
25
26         @Size(min=3, max=20, message=
27            "Username must be between 3 and 20 characters long."
                 )
28      @Pattern(regexp="^[a-zA-Z0-9\\s]+$",
29       message="Username must be alphanumeric")
30      private String username;
31
32      @Pattern(regexp="[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z
            ]{2,4}",
33              message="Invalid email address.")
34      private String email;
35
36      @Size(min=6, max=20, message="Passwords must be between 6
            and 20 characters long.")
37      @Pattern(regexp="^[a-zA-Z0-9\\s]+$",
38          message="Passwords must be alphanumeric")
39      private String password;
40
41      .... // other fields
42
43      }
```

Behind the scenes, SpringMVC will check that the parameters entered in the html form are valid in terms of the requirements specified in the Shredder class. If there is an error, the BindingResult instance in my controller function would have the hasErrors() function return true, in which cause I simply return an error message back to the user. Using

regular expressions gives me the power to decline all input forms that does not match the given pattern.

Before I went with this approach, I wanted to try another solution using negative filtering. In this implementation I created a custom validator implementation that would perform negative filtering on the user data. The Validator is just a custom specific java implementation that is executed before the controller handler for a url is run [58]. In my validator class, I used the builder design pattern to make sure all the required fields where filled out properly, and if any potential characters or patterns were found (like for instance the sequence " ' OR 1=1; SELECT ...; – ", I would return an error message to the controller function. The builder pattern provided a coherent and intuitive implementation of Shredder validation. But as already mentioned, implementing negative filtering is a difficult approach, and besides, if I was to add one builder class for every possible form validation function, I would end up having a big pile of classes.

Displaying model data To be able to populate the html pages with data stored on the server, I have to use a template scripting language as explained in chapter 2. As already mentioned, SpringMVC comes with a template engine that renders jsp views into an html page using data in the model object. A good choice is to use JSP, because it has support for lots of tag libraries (like JSTL), and it has a nice and intuitive syntax, that is more java-like then for instance Velocity or Freemarker. However, some could argue that choosing Velocity or Freemarker is a better approach because its syntax is more appealing to non-java developers. The goal of my jsp implementation is to make it as easy as possible to integrate my domain objects (shredders and shreds) into the html pages. Also, it is best practice to avoid implementing business logic in the jsp pages, as this should be done server-side **??** on page ??. View logic however, should only be done in the view layer, or else I would get a tight coupling between the view layer, and the model layer. The user interface has some challenging requirements when it comes to dynamic user-interaction, so obviously the template language should work well with both javascript, css and html. Below is an example of how a list of shreds is iterated over and displayed (Notice I have omitted irrelevant tags as this is not important for this example):

```
1   <c:forEach var="shred" items="${shreds}">
2         <!—— Set the shred rating, avoid divide−by−zero error
              —>
3         <c:choose>
4           <c:when test='${shred.rating.numberOfRaters > "0"}'>
5             <c:set var="rate"
6               value="${shred.rating.currentRating/shred.rating
                  .numberOfRaters}" />
7           </c:when>
8           <c:otherwise>
9             <c:set var="rate" value="0" />
10          </c:otherwise>
11        </c:choose>
12          <!—— Shred video —>
```

```
13          <video id="shredVideo" class="video−js vjs−default
              −skin" controls>
14            <source src="<c:url value="/resources/images/"/>
              ${shred.videoPath}" type='video/mp4'>
15          </video>
16          <!−− Rating −−>
17          <form action="<c:url value='/shred/'/>${shred.id}?
              action=rate"
18            method="POST">
19            Rating: ${rate}. Rate it: <input type="range"
                min="0" max="10"
20              name="rating" value="5"
21              onchange="showNewRateValue(this.value, ${
                shred.id})" /> <span
22              id="range${shred.id}">5</span>
23            <button type="submit" class="btn btn−success
                btn−small">Ok</button>
24          </form>
25          <p class="lead">${shred.description}</p>
26            By: <a href="<c:url value='/shredder/'/>${shred.
              owner.id}">${shred.owner.username}</a>
27          <p>Created at: ${shred.timeCreated}</p>
28            Tags:
29            <c:forEach var="tag" items="${shred.tags}">
30              <span class="label label−info"> <a href="${tag
                .label}"
31                style="color: white">${tag.label}</a>
32              </span>
33            </c:forEach>
34          <!−− comments >
35          <a class="btn" href="#" onclick="openCommentBox(${
              shred.id});">Comments</a>

37          <div id="commentBox${shred.id}" hidden="true">
                                        <c:forEach
              var="comment" items="${shred.shredComments}">
38            <!−− Here comes comments. This is omitted for
                this example −−>
39            </c:forEach>

41          <!−− post a comment −−>
42            <form id="comment"
43              action="<c:url value='/shred/${shred.id}/
                comment/'/>"
44            </form>
45        <!−− End shred for >
46        </c:forEach>

48    <script type="text/javascript">
49      function openCommentBox(id) {
50        var tableId="#commentBox" + id;
51        if ( $(tableId).is(":visible")) {
52          $(tableId).hide();
53        } else {
54          $(tableId).show();
55        }
56      }
57      function showNewRateValue(newValue, postfixId)
58      {
```

```
59        var  id  =  "range"  +  postfixId ;
60        document . getElementById ( id ) . innerHTML=newValue ;
61    }
62  </script>
```

The jsp page uses HTML 5 tags like video and range (used to display a rating bar for each video). These features makes it easy to build the main gui-components that I need in the application. Videos are simply fetched from the a url that lies on the server and is being displayed through a javascript tag on the html page. Another approach is to use streaming technology, where raw bytes are transferred in real time while the user watches the shred. However, the performance benefits with choosing either of these approaches is out of scope for this project, so I won't dig into it.

A big problem with the template approach is that there is a lot of mixed responsibilities and languages. The for loop itself is very big, and contains both html, javascript and jsp logic. Imagine having to add more advanced features that requires more use of jsp references mixed with javascript handling. The source code for the whole page as it is is 254 lines of code, plus 146 lines that are included from other general jsp pages, like the header and side navigation. And this is not including the external javascript and css libraries! It is also important to notice the big amount of code that is generated for a simple if-else condition in the top of the source. It only gets worse when more logic like this has to be implemented. Lets show how ugly this can get when the jsp tags must perform more view logic. This shows part of the code that generates a battle page (again, I have omitted (lots of) irrelevant markup code):

```
1   <!-- Display who's the leader -->
2   <c:choose>
3     <c:when test='${battle.battler.battlePoints > battle.battlee.
          battlePoints}'>
4       <p> ${battle.battler.shredder.username} Leads </p>
5     </c:when>
6     <c:when test='${battle.battlee.battlePoints > battle.battler.
          battlePoints}'>
7       <p> ${battle.battlee.shredder.username} Leads </p>
8     </c:when>
9     <c:otherwise>
10      <p> It's even </p>
11    </c:otherwise>
12  </c:choose>
13
14  <c:forEach var="shred" items="${battle.battler.shreds}">
15      <legend>Round ${shred.round}</legend>
16      <!--Display a shred like in the last example, including
           video, rating, comments and shred details -->
17  </c:forEach>
18
19      <!-- If the currently logged in shredder is next up to shred
           , -->
20      <!-- And he is the battler in the battle, meaning he is on
           the left side of the screen -->
21      <!-- Then display a button to upload a shred -->
22      <c:if test="${(battle.currentBattler.id == shredder.id) && (
           battle.battler.shredder.id == shredder.id)}">
```

```
23
24        <a href="#battleModal" role="button">Shred back!</a>
25        <!--- Battle modal --->
26        <div class="modal hide fade" id="battleModal" aria-hidden=
             "true">
27            <button type="button" class="close" data-dismiss="
                 modal"
28            aria-hidden="true">x</button>
29            <h3 id="myModalLabel">Battle shredder</h3>
30          <!--- video upload --->
31          <form method="POST" class="form-horizontal"
32          enctype="multipart/form-data"
33
34            <!--- This programmer unfriendly piece of code
                 statement builds a url that triggers the
                 controller handler for uploading a shred in a
                 battle --->
35            action="<c:url value='/battle'/>/${battle.id}/shred/${
                 battle.currentBattler.id}/fn:length(battle.battler
                 .shreds) + 1}">
36            <label class="control-label" for="shredVideo">Shred
                 video</label>
37            <input name="shredVideo" type="file" placeholder="
                 Browse" />
38
39            <!--- Hidden field that must be added to the http
                 request.. Not easy to spot this one! --->
40            <input type="hidden" name="shredderId" value="${
                 shredder.id}">
41            <button class="btn" data-dismiss="modal" aria-hidden="
                 true">Close</button>
42            <button class="btn btn-primary">Enter</button>
43          </form>
44        </c:if>
45
46        <!--- For all shreds that belongs to the battlee, on the
               right hand side --->
47        <c:forEach var="shred" items="${battle.battlee.shreds}">
48          <!--- Do the EXACT same thing as showed above, but for
               the left hand side shredder (battlee)--->
49        </c:forEach>
50
51      <!--- If the currently logged in shredder is next up to shred
             , --->
52      <!--- And he is the battlee in the battle, meaning he is on
             the right side of the screen --->
53      <!--- Then display a button to upload a shred --->
54      <c:if test="${(battle.currentBattler.id == shredder.id) && (
             battle.battlee.shredder.id == shredder.id)}">
55        <!--- Do the exact same thing as showed above, but for the
               left hand side shredder (battlee)--->
56      </c:if>
```

Obviously, this code is not pretty and contains a lot of duplication. The code that implements the battler's shreds is duplicated with a few variations on the right hand side, for the battlee. It is hard to avoid this when I am restricted to jsp tags, and plain html. Again, the jsp if conditions is not very pretty. If I was to add more features, I would probably have

to extend the code with more duplication, as it is hard to create reusable components with this technology. All in all, the implementation violates both the DRY principle, and the SRP principle. Another issue is that the jsp statements tend to violate the law of demeter [48], by accessing a particular objet through a hierarchy of pointer references. For instance: *<c:if test="${(battle.currentBattler.id == shredder.id) & & (battle.battler.shredder.id == shredder.id)}">*. Using such references makes the code more error prone, should for instance the data structure in my domain layer change. Such references are difficult to avoid in jsp, because they cannot access server-side functions, only variables (through getter-methods). Unfortunately, I am using such referencing in various parts of my app, but I try to avoid it.

I want to pinpoint out one last example of how "smelly" the front-end code gets with my template approach. This example shows the ShredNews implementation. Each time a shred-newsItem is hovered, it should display a box with type-specific content.

```
1   <script type="text/javascript">
2
3   function dopopover(id, description, content ) {
4     $('#shred-'+id).popover(data={
5       trigger:'hover',
6       placement: 'left',
7       content: content,
8       title: description,
9       animation:true
10    });
11  }
12
13  function createShredContent(tags, rating) {
14    tags = tags.replace(/,\s*$/, "");
15    return "<p>Tags: " + tags + "</p>" + "<p> Rating: " + rating +
          "</p>";
16  }
17
18  function createBattleContent(round, status, shredder, shreddee)
        {
19    var str = "<p>"+shredder + " vs. " + shreddee + "</p>";
20    str += "<p>Round " + round + "</p>";
21    str += "<p>Status: " + status +"</p>";
22    return str;
23  }
24  function createShredderContent(adress, description, guitars) {
25    guitars = guitars.replace(/,\s*$/, "");
26    var str = "<p>Lives in: " + adress +"</p>";
27    str += "<p>"+description +"</p>";
28    str += "<p>Shreds on: "+guitars+"</p>";
29    return str;
30  }
31  </script>
32
33  <div class="span3">
34    <div class="well sidebar-nav">
35      <ul class="nav nav-list">
36        <li class="nav-header">Shred news</li>
37
```

```
38          <c:forEach var="newsitem" items ="${news}" varStatus="
                status">
39          <%
40          ShredNewsItem item = ((ShredNewsItem) pageContext.
                getAttribute("newsitem"));
41          if (item instanceof NewShredFromFanee ){
42          %>
43
44          <!-- Create the list of tags for the shred -->
45          <c:set var="tagsList" value=""></c:set>
46          <c:forEach var="tag" items="${newsitem.shred.tags}">
47            <c:set var="tagsList" value="${tag.label}, ${tagsList}"
                />
48          </c:forEach>
49
50          <li>
51          <a rel="popover"
52            id="shred-${status.count}"
53            onmouseover="dopopover(${status.count},
54            '${newsitem.shred.description}',
55            createShredContent('${tagsList}', ${newsitem.shred.
                rating.rating}));"
56            href="<c:url value='/shredder//'/>${newsitem.shred.owner.
                id}">
57            ${newsitem.shred.owner.username} just added a new shred
                at ${newsitem.timeCreated}</a>
58          </li>
59
60          <%
61          } else if (item instanceof BattleShredNewsItem ) {
62          %>
63          <li>
64          <a
65          id="shred-${status.count}"
66          onmouseover="dopopover(${status.count},
67          'Battle ','${newsitem.battleShred.owner.username} shredded
                in round ${newsitem.battleShred.round}');"
68          href="<c:url value='/battle//'/>${newsitem.battleId}">${
                newsitem.battleShred.owner.username}
69           shredded in a battle at ${newsitem.timeCreated}</a></li>
70
71          <%
72          } else if (item instanceof NewBattleCreatedNewsItem ) {
73          %>
74          <li>
75          <a
76          id="shred-${status.count}"
77          onmouseover="dopopover(${status.count},
78            'Battle ',
79            createBattleContent(${newsitem.battle.round},'${newsitem
                .battle.status}', '${newsitem.battle.battler.
                shredder.username}', '${newsitem.battle.battlee.
                shredder.username}'));"
80           href="<c:url value='/battle//'/>${newsitem.battle.id}">${
                newsitem.battle.battler.shredder.username}
81           just entered a Battle against ${newsitem.battle.battlee.
                shredder.username} at ${newsitem.timeCreated}</a></li
                >
82
```

```
83        <%
84          } else if (item instanceof NewPotentialFaneeNewsItem ) {
85        %>
86        <!-- Create the list of guitars. Don't bother doing
               equiptment, it's such a hazzle! -->
87        <c:set var="guitars" value=""></c:set>
88        <c:forEach var="g" items="${newsitem.shredder.guitars}">
89          <c:set var="guitars" value="${g}, ${guitars}" />
90        </c:forEach>
91        <li>
92        <a
93        id="shred-${status.count}"
94        onmouseover="dopopover(${status.count},
95          '${newsitem.shredder.username}',
96          createShredderContent('${newsitem.shredder.adress}','${
               newsitem.shredder.description}','${guitars}' ));"
97         href="<c:url value='/shredder/'/>${newsitem.shredder.id}"
               >${newsitem.shredder.username} just joined ShredHub.
98        You are shred-alike. Check out his shreds.</a></li>
99        <% } %>
100       </c:forEach>
```

The code absolutely violates the single-responsibility-principle in that it has a big set of if-elses for each type of news item. If I were to add more news items I would have to add another else clause. This is only intentional, because I don't want to able to do something like ShredNewsItem.htmlView that uses inheritance to create a concrete shredNewsItem view implementation as an html generated java String. The reason is as I have already mentioned, back-end code should have no clue of how it is displayed in the front-end. Therefore, I have to check the type of the java object, and create 4 customized views in the jsp page. Obviously this gets very ugly, because I have to use both jsp-scriptlets, jstl tags and javascript in the same pile of code. Now, it may be that there are ways to do this with a cleaner and more concise implementation, but it will never save me from the problem with using template languages; it has to be rendered on the server, so the code is not accessible from javascript statements on the browser. This creates a difficult cooperation between jsp and javascript, and special "hacky" solutions has to be done to solve minor implementation details.

Authentication Users must to be authenticated to access the system. Before they can start using the web application, they must register themselves and create a password. This process can be implemented fairly easy by using the Spring security framework [59]. Since security is not a big part of this project, I will just give a short description of my implementation. First of all, shredders are stored in the database with a username, password and security token. The token is the same for every shredder, which means they have access to everything on the web app. I could create more complicated authorization levels, say for instance if I wanted to protect certain urls with a higher security requirements. A spring security configuration file in my application is set up to require all urls except the login page to require a logged in user with the right

security token. Spring will therefore create interception filters that verifies the user each time he tries to access a url. Here is a snippet of my security configuration file:

```
1      <intercept-url pattern="/theShredPool*" access="
          ROLE_SHREDDER" />
2      <intercept-url pattern="/battle*" access="ROLE_SHREDDER" />
3      <intercept-url pattern="/battle/**" access="ROLE_SHREDDER" /
          >
4      <intercept-url pattern="/shredder*" access="ROLE_SHREDDER" /
          >
5      <intercept-url pattern="/shredder/**" access="ROLE_SHREDDER"
           />
6
7        <authentication-manager>
8          <authentication-provider>
9          <jdbc-user-service data-source-ref="dataSource"
10        users-by-username-query="
11           select Username, Password, true
12           from Shredder where Username=?"
13
14        authorities-by-username-query="
15           select s.Username, ur.Authority from Shredder s,
              UserRole ur
16           where s.Id = ur.ShredderId and s.Username =?  "
17      />
18        </authentication-provider>
19    </authentication-manager>
```

Session management Another important feature of the application is session management. Since http is stateless, the server has to maintain state between a given client's requests. A session is a term that defines the data (state) associated with a user when he navigates around on a given web page. Obviously users don't want to have to log in to the system every time a new request is made. Now, there are a couple of ways to implement sessions. One can store the state in cookies, add a session id to each url, or inside hidden fields in html forms. But the most obvious choice is to use Html session objects. This is implemented as a session-token identifier that is stored either in a cookie or in the url (by using url rewriting). Java servlet technology implements this inside the javax.servlet.http.HttpSession interface, which is wrapped by the Spring MVC framework. Therefore, I will use this technique to maintain state in my application. An example of an http request where the session id is stored in a cookie is given in figure 5.2 on the facing page. Notice the session id is given in the field *JSESSIONID*.

The application very often needs to know the identity of the currently logged in user. In addition to user authentication for each http request, the app needs user information to be able to perform correct lookups in the database when a request is made for something. For instance, if the user asks for a list of shredders he might know, or the user asks for a list of all his fanees. This is also the case for post requests, like if the user wants to add a shred, or to comment or rate a shred etc. The application

```
Request URL: http://localhost:8080/shredhub/shred/recommendations?
action=fanShreds&shredderId=7
Request Method: GET
Status Code: ● 200 OK
▼ Request Headers      view source
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Connection: keep-alive
Cookie: JSESSIONID=9620143285B9E81EBD15D4DC938918E6; openid_provider=google
Host: localhost:8080
Referer: http://localhost:8080/shredhub/shredpool
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/537.4 (KHTML, like Gecko
) Chrome/22.0.1229.94 Safari/537.4
▼ Query String Parameters      view URL encoded
action: fanShreds
shredderId: 7
```

Figure 5.2: A HTTP get request for a list shreds made by fanees

would in these cases have to have the id for the currently logged in user. In addition, imagine if the user wants to receive a list of new shredders who might be of interest to him, the application would then need to know the shredder's home address, and the list of his guitars and musical equipment. The point is, the application definitely needs to know the details of the user when requests are made to the server. Therefore, the shredder object should be kept in memory, otherwise the currently logged in user would have to be fetched from the database almost each time a server request was made. So obviously, the session object should be used as a performance cache to avoid the tedious database lookups. Now, it is interesting to investigate if there are other data objects that should be kept in memory too, so that it won't need to be fetched from the database each time a request for it is made. Now I don't want to focus too much on getting the perfect performance using the specific architecture I have laid out here, because this thesis is about exploring other ways of building this web app. But to be able to do good comparison, I will do my best to create a best-practises solution for each architecture I propose. So to try and achieve good performance with this architecture without violating the need for real-time updates I will try and define a clear policy for using the session object as a server-side cache store.

Given below I have categorized 3 different lists of data objects that are requested in the application, together with an indication of how frequently the object changes, how often it is used, and the conclusion if they should be cached in the HTTP session object.

1. The list of the objects that always stays visible in the app:

Data object	Changes	Used	Stored in session
Username of the currently logged in user	Seldom	Always	Yes
The list of available shred-tags	Seldom	Always	Yes
The list of ongoing battles for the user	Moderate	Always	Yes
The list of requested battles for the user	Often	Always	Yes
A list of the user's fanees	Moderate	Always	Yes
Id for the currently logged in user	Never	Very often	Yes

2. The list of other data objects that are requested in the app

Data object	Changes	Used	Stored in session
User details, like address, guitar and equiptment list etc	Seldom	Moderate	Yes
A list of shreds based on the users profile	very often	Often	No
A list of shreds with no attachment to the user, for example, top shreds, shreds with certain tags etc	Very-often	Often	No
A list of the shreds made by the user	Moderate	Moderate	No
The list of other shredders (non-fanees)	Often	Often	No

3. The ShredNews list

Data object	Changes	Used	Stored in session
A fanee entered a shred battle	Often	Often	No
A fanee shredded in a battle	Often	Often	No
A fanee posted a new shred	Very often	Often	No
A potential fanee started using Shredhub	Often	Often	No

Good caching techniques states that objects that are frequently accessed should be cached, and objects that are rarely used should stay in the

database. But, if the object frequently changes (like shreds, battles and shredders), they should not be cached even if they are accessed frequently. This is specific to my application, because with a lot of simultaneous users, one would want to restrict the amount of memory used on the server as much as possible to avoid slow performance or out of memory exceptions. Also, considering shreds are created very often, they should be written to the database immediately (direct cache) when they are created, instead of being temporarily stored in memory. This is obvious, because users want real-time updates when someone creates a new shred. This is the motivation for the decision I have stated in the above listing. The logged in user should stay on the session object, together with the list of battles he participates in, and the list of fanees and tags. Now it might be that these datas will change (for instance that a new battle request comes in). In this case there are some alternatives to choose. First of all, objects that the user can change himself (like profile details and adding new fanees), is just a matter of updating the session object server-side and write to the database each time this happens. The challenge comes to when external events happens, like if someone challenges the user for a new battle, or someone accepts a battle the user has initiated. In this scenario it is preferable for the user to get the update as quickly as possible. The easiest solution for this is to not update the session object at all, in which case the user would only get the newest updated data each time he logs into the app. Another solution is to update the session every time the user enters the shred pool (naturally, since the shred pool acts as the "home area" in the web app). In which case the update will not be exactly real time (depending on how the user uses the app of course), but probably good enough. A third solution is to use the observer pattern by implementing a thread that propagates the update to the designated session objects when an event like this happens. This is called a "push" model and is a bit more complex to implement. A final solution is that the client "polls" for updates in a particular time interval. Now, my architecture is built around a model where the server always responds with a full html page. If the client wants to receive only a particular data object (for instance a new battle), he should not have to receive a full html and see the whole page render each and every x seconds. A preferable solution is to make ajax polls or something similar which I will look into in another architecture I propose later in this project. Therefore, at this point I will confine myself with option 1.

As for the objects that are not persisted on the session object, they will be updated every time the page is refreshed. A pitfall with this is that if the user stays on the same page for a long time, he will not receive any updates. A dynamic behavior is preferable (as stated in the application goals earlier), which would require either some form of push or pull model. I described this scenario in the section above, so again I will state that the manual update model is good enough for this architecture, and state that a better solution will be implemented in the architecture I will look at later in this project.

Identifying advantages with the web tier architecture

In this section I will discuss the advantages with this web tier architecture.

Separation of concerns   There is a clear separation between the controller layer and the middle layer in my architecture. This is desirable, because if I was to change the web tier to say for instance, a Struts implementation, the service (or model) layer would not be affected, because it has no dependencies back to the presentation tier. Neither does the presentation tier ever touch the data tier.   This responsibility is managed by the middle layer.All business functions are delegated through references to service objects (like ShredderService as seen in the code above), which are composed into the controller classes by dependency injection [**ioc**]. Hence, the web tier completely conforms to the dependency inversion principle [**dip**], as well as the Single-responsibility principle [**dip**].

Simple architeture   Obviously this is a classic implementation of a three layered architecture, so it should be fairly easy for new developers to get a grasp of the whole system, especially considering well-known languages like java and jsp are being used.   The controller functions has a nice mapping to urls, making it easy to create new controller functions (actions).

Easy to test   The controller classes are simple java classes that are easy to test. Spring provides a simple mechanism for mocking http requests, as well as service objects. That way I can create unit tests that does not require difficult configuration setups.

Identifying limitations and drawbacks with the web tier architecture

In this section I will discuss the limitations and drawbacks with the web tier architecture.

Platform-dependency   A big problem with this approach is that the server always responds with html pages when a request comes in. This is unfortunate, because clients might be third-party applications that just want to display some data fetched from my application, like a list of top shredders. Therefore, the application should support other formats then just html.   This would also make it easier to implement polling functionality, since the client could ask the server for a data object represented as an xml or json object.

Tight coupling   The jsp pages are tightly coupled with the domain objects and the html/javascript code. This is unfortunate if I was to change my domain model, or add complex javascript logic that depends on the data returned by jsp tags. Making a clean collaboration between jsp tags, javascript and a java backend is difficult, and becomes even worse the more complex the application gets.

Cluttered front-end code The jsp pages contains code that is untidy, has lots of duplication and no object abstractions. This makes it hard to build a nice and clean front-end implementation, because there is no clear structure in the code. As showed in the examples, there is a lot of front-end logic, which should be encapsulated in functions and classes. I have acknowledged that there is potential for an improved jsp implementation, but it does not involve solving the problem of mixing jsp tags, javascript, html and css, or providing a clean, object oriented front-end implementation.

Needless amounts of data between client and server Each time a request comes in, the server returns ha complete html page back. For example, when a shredder is in the shred pool, and wants to change the list of shredders he sees, to for instance get the top list of shreds, not only is the new list of shreds returned, but also everything else that surrounds the list! That could be a lot of data. Another example is when a shredder uploads i video, the video is sent to the server, and the browser would receive a url redirect, meaning a new http request has to be made to the server, so that a new html page can finally be returned back to the browser. Thats 2 html request/response pairs with unnecessary data.

Manual page refresh When a shredder wants to fetch the newest data, he has to update the page to actively send a server request. A more preferable approach is to have this process happen automatically without having the user ever notice it (except that when a new battle request, or shred video pops up of course).

Data stored on the session object There is some amounts of data that are stored on the session object, like fanees, battles, tags and user details. Clearly when the amount of logged in users are high, there is a lot of data in the memory. This can lead to slow performance and in the worst case out of memory exceptions. Also, the data that is stored on the session might become outdated during a user session, because I only update the session object for events that are made by the user, and not for external events. This behavior is cumbersome, and an improvement is challenging.

5.4   The middle tier

The middle tier 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. There is exactly one service function for each controller handler. Now, since the responsibility of the middle tier is to perform the business logic in the application, this tier should be very flexible. Flexibility, means that it should be easy to add new features without harming anything else in the source code, and it should be easy to modify already existing code. To be able to get this behavior, one needs
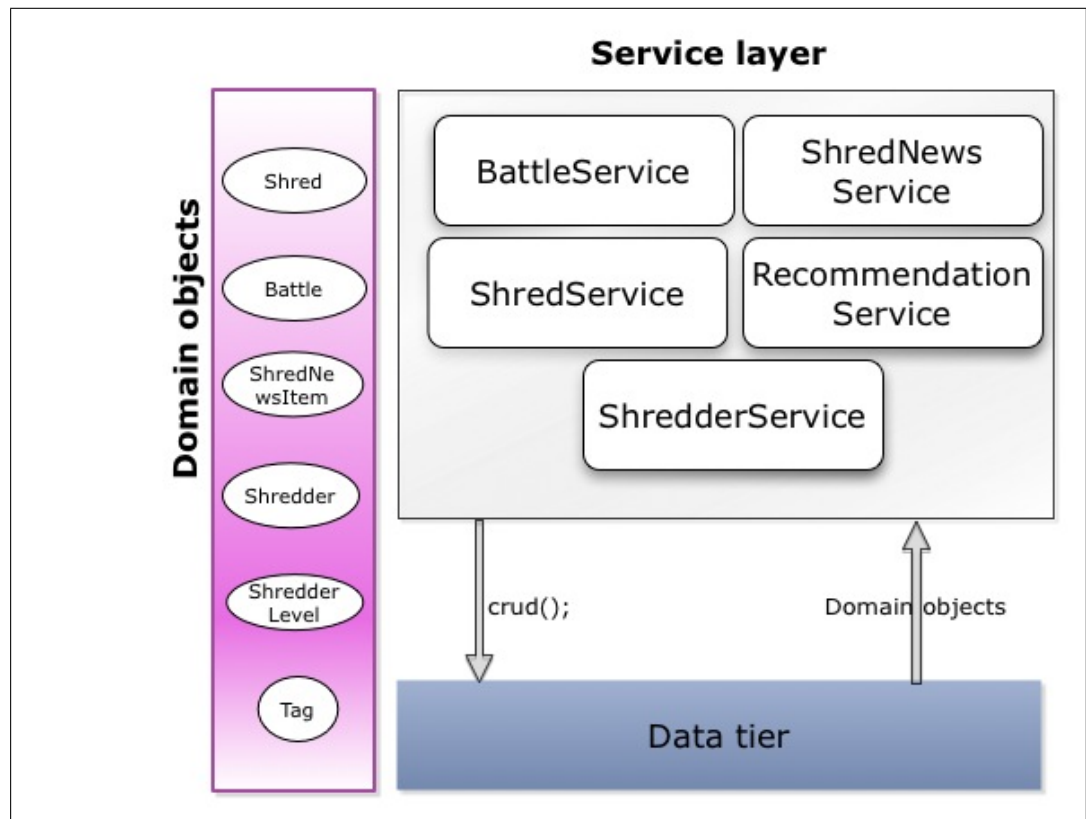
Figure 5.3: The middle tier and its connection to the data tier

a coherent design, preferably built with good design patterns that fits into the whole architecture. To organize the middle tier, 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 is represented as a service class, which has the service functions that each abstraction should provide. The list of service classes with some essential functions are given below:

1. BattleService (getBattleWithId, getOngoingBattlesForShredderWithId, acceptBattleWithId)

2. RecommendationService (getRecsBasedOnTags, getRecsBasedOn-Ratings, getRecsBasedOnFansOfShredder)

3. ShredderService (addShredder, getShredderWithId)

4. ShredNewsService (getLatestShredNewsItems)

5. ShredService (getFanShreds, getAllTags, getShredsForShredderWithId)

An overview of the middle tier, and its responsibility according to the data tier is given in figure 5.3. The domain objects are simple POJO's with no business logic functionality.

The responsibility and hence the supported operations given for each service class should be obvious. Now, I could have chosen to organize the middle tier using another pattern. One alternative is the table module pattern [55], where all 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, where the domain objects (Shredder, Shred, Battle etc) are responsible for performing the business logic, and talking to the database. This is a good approach, but the domain objects end up being very big with lots of responsibility. I prefer the solution of having a service class for each logical application abstraction, where each service function performs all the necessary business logic for a given action. The domain objects are simple POJO's without any business functions. However, letting the domain objects perform certain business operations that are domain specific is a possibility, but I have chosen to have a clean POJO representation of the objects, and instead implement domain specific operations in separate classes that can be reused across the service classes.

A final thing to point out is that since some service operations performs multiple subsequent database transactions, each such service operation must be atomic. Spring achieves this behavior very elegant by letting classes be declared as transactional (as described in chapter 2), in which case each operation in the given class would be performed in atomic matters. An example of a service class is showed in the code below.

```
1   @Service
2   @Transactional( readOnly=true )
3   public class ShredServiceImpl implements ShredService {
4
5     // Data tier reference
6     @Autowired
7     private ShredDAO shredDAO;
8
9
10    /**
11     * When a shred is rated, the shred will gain a higher total
             rating, and
12     * the shredder who made the shred achieves
13     * more experience points
14     */
15    @Transactional ( readOnly = false )
16    public void rateShred(int shredId, int newRate) throws
             IllegalShredArgumentException {
17      Shred shred = shredDAO.getShredById(shredId);
18      if ( shred == null ) {
19        throw new IllegalShredArgumentException("Shred with id: "
               + shredId + " does not exist");
20      }
21      ShredRating currentRating = shred.getRating();
22
23      // Here I could use the domain model pattern so that the
24      // current rating knows how to set its own rating.
25      // But I might as well do it here, because it is nice to
             only
```

```
26        // operate  with  pojos
27        currentRating.setNumberOfRaters(currentRating.
             getNumberOfRaters() + 1);
28        currentRating.setCurrentRating(currentRating.
             getCurrentRating()+newRate);
29
30        shredDAO.persistRate(shredId, currentRating);
31
32        // Update  xperiencepoints  for  shredder
33        Shredder shredder = shredderDAO.getShredderById(shred.
             getOwner().getId());
34
35        // Uses  a  utility  class  that  is  shared  by  all  the  service
             classes
36        // The  rating  rule  is  super  simple. I  should  come  up  with
             something  more  killer..
37        UpdateShredderLevel usl = new UpdateShredderLevel(shredder,
             newRate);
38        usl.advanceXp();
39
40        shredderDAO.persistShredder( shredder);
41    }
42  }
```

Identifying advantages with the middle tier architecture

The middle tier has a nice and coherent implementation. It has a clear separation from the presentation tier and the database tier, which makes it easy to swap out any of these tiers. Each service class represents a logical abstraction in the application, which makes the design very intuitive.

Identifying limitations and drawbacks with the middle tier architecture

It really isn't that much too say about the limitations with this tier. However, some of the service classes are quite large and it could be preferable to divide these classes into smaller ones to avoid too much responsibility.

5.5 The data tier

The data tier is the part of the application that receives a particular CRUD command from the service tier, executes a sql statement on the database, maps the result to POJO's and returns the result back to the service tier. I have chosen to use a Postgres implementation, simply because I prefer to build my data tier using sql statements. There are other object relational mapping technologies like hibernate, and JPA, which gives the developer the freedom to operate only on the object level, not having to worry about 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 postgres' proprietary features. Such flexibility and fine-tuning does become more difficult when one is limited by an ORM barrier like hibernate.

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. My application is built around two central tables, namely Shreds and Shredders. The tables for these are given 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  );
14  CREATE TABLE Shred (
15      Id            serial PRIMARY KEY,
16      Description     text ,
17      Owner        serial REFERENCES Shredder(Id),
18      TimeCreated    timestamp DEFAULT CURRENT_TIMESTAMP,
19      VideoPath     varchar(100) NOT NULL,
20      ShredType     varchar(30) DEFAULT 'normal' CHECK (ShredType ='
            normal' or ShredType = 'battle')
21  );
22  CREATE TABLE Battle (
23      Id            serial PRIMARY KEY,
24      Shredder1     serial REFERENCES Shredder(Id),
25      Shredder2     serial REFERENCES Shredder(Id),
26      TimeCreated    timestamp DEFAULT CURRENT_TIMESTAMP,
27      BattleCategori     serial REFERENCES BattleCategori ,
28      Round        int DEFAULT 1,
29      Status        varchar(30) DEFAULT 'awaiting' CHECK (Status ='
            accepted' or Status ='declined' or Status='awaiting');
30  );
```

Now, there are lots of other smaller tables then these as well, but these two are the essential ones. To perform the CRUD operations, I have chosen to structure my data tier around the Data access object pattern. In this pattern, separate DAO objects are responsible for performing the relational data mapping on behalf of the domain objects. 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 5.4 on the following page shows how the object relational
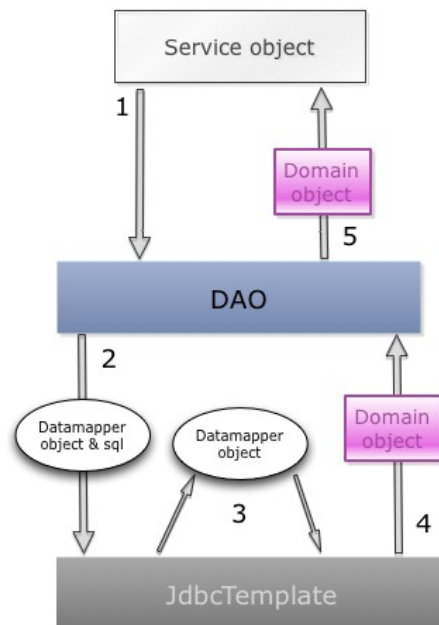
Figure 5.4: Data access pattern in the data tier

mapping is done in the data tier. Heres what happens when the middle tier asks the data tier for some dabase object (notice that the service object might do multiple subsequent calls to the data tiers in one atomic transaction.

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

2. The DAO class uses a JDBCTemplate instance (provided by spring) to delegate the boilerplate behavior like getting the database connection and handling exceptions. The JDBCTemplate is also responsible for executing the database query itself, provided that it gets a SQL statement from its caller. For instance:

```
@Service
public class ShredDAOImpl implements ShredDAO {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public Shred getShredById(int shredId) {
        String sql = "SELECT * FROM Shred s,Shredder sr WHERE
            s.Owner = sr.Id AND s.Id = ?";
        return jdbcTemplate.queryForObject(sql, new Object[]{
            shredId}, new ShredMapper());
    }
}
```

3. The jdbctemplate is built with the template design pattern, meaning it does callbacks to the mapper object provided by the caller. In

the above code, this would be the ShredMapper instance, which is responsible for marshaling the result from the sql query. I created these mapper objects so that they can be reused across all the dao objects. Example:

```
1   public class ShredMapper implements RowMapper <Shred >{
2
3     public Shred mapRow(ResultSet rs , int rowNum) throws
          SQLException {
4       // Template method!
5       Shred shred = this.setConcreteShredder();
6       shred.setId(rs.getInt("id"));
7       shred.setDescription(rs.getString("Description"));
8       shred.setOwner( new ShredderMapper().mapRow(rs , rowNum)
          );
9       // Another template method!
10      this.addConcreteBehavior(rs , shred , rowNum);
11      return shred;
12    }
13  }
```

Notice that I also use the template method pattern to enable customized mapper functionality, like BattleShredMapper who is a subclass of ShredMapper, and will therefore provide concrete implementations of the specific template methods.

4. The domain object 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 object. Notice that the exceptions JDBCTemplate might throw are unchecked exceptions, so I only catch an exception if it's necessary for the client user.

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

Handling media content

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 has a maximum size of (x) bytes. 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 client tier section, videos are downloaded to the clients web browser before it is displayed. 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 access which ever files they want, 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 a free service up to some x amount of requests and x amounts of data. In the section on architecture 2.0 I will show how this was done. There are other alternatives to amazon, like SimpleCDN and windows azure, and they all look very similar in both pricing and performance.

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 create a Shred object, a Shredder (owner), all the tag objects for the shred, all the comments etc. One approach is to fetch everything, which would require some amount of processing and some unnecessary data returned back to the client in the browser. Another approach is to implement lazy loading techniques in which case only the most important data is populated in the domain object(s), and when a jsp page tries to access a field that is not yet fetched from the database, the field will be fetched on demand. However, this requires a different architecture, because suddenly the domain objects (who are the ones that are being referenced from the jsp page) would either have to implement the virtual proxy pattern and would have to know how to access the DAO objects, which is really the responsibility of the service objects. Now, I have implemented a solution for this, in which case the JSP pages does not access the domain objects directly, but instead goes through a Identity Map [**imap**]. The problem with this though, is that it has to store a lot of objects in the servers memory, which is unacceptable with this high demand for many simultaneous users. Therefore, I chose to simply fetch just enough fields that are required by the client caller. This is a simple and stateless approach, and it might give performance benefits if the client user does not follow a chain of connected objects (which would lead to lots of fetching of the same data objects). On the other hand, it might result in lots of duplicated database reads. Also, it does put a certain amount of coupling between the service operations and the CRUD operations, but its worth the tradeoff, as long as I am conscious.

Identifying limitations and drawbacks with the data tier architecture

One drawback is that the DAO objects tend to get really big with lots of functions. However, considering the amount of queries the application requires, this is probably unavoidable. Another thing to take into consideration is that the DAO pattern does provide an additional layer of complexity. As mentioned earlier the active record pattern avoids this,

since the domain objects knows how to "crud" themselves, but I concluded that the tradeoff of having complex domain objects was not worth it. Also, its a big challenge to define how much is the optimal amount of data that should be fetched in the various queries when the server should not maintain any in memory objects (making lazy loading more difficult). Obviously, some sort of caching that does not use the server's memory capacity would be preferable. A final remark is that considering all the available non-sql technologies that has recently hit the marked, I have chosen to see if there is a potential for improvement in terms of efficiency, scalability and a cleaner architecture, by swapping out my data tier with a no-sql data tier.

Identifying limitations and drawbacks with the data tier architecture

The DAO pattern provides a coherent data object relational mapping implementation that is nicely decoupled from the service layer and the domain objects. Using sql directly in java tend to be cluttering, but it does give me a lot of query flexibility. Debugging is definitely easier then it would be using an ORM implementation like hibernate. Really, the advantage of having full control is so much better then being stuck behind some massive complex ORM framework.

5.5.1   Discussion

In this section I will wrap up the most important fallacies and pitfalls with architecture 1.0, and propose an improvement that I will choose to implement for architecture 2.0. Below I have categorized the primary weaknesses in the application.

Front-end code
The front end code is implemented by mixing many different technologies like JSP, scriptlets, javascript, html and css. Considering the high requirement for a dynamic user interface, I should have had a clean, flexible and structured front-end implementation, with separate concerns, no language mixes, and no 500+ lines of source code for each html page.

Html response
One complete html page is returned for each and every http request from the user. This is definitely overkill, considering the client might even be staying on the same page, but just wants to get a different set of shred videos. The result slow feedback and waste of bandwidth.

No support for 3rd party client applications or mobile clients 3rd party clients and mobile client application's prefer another format then html when using my application. With a simple format like xml or json, and the data types being application domain objects instead of html pages, other applications could find my application useful, and I could easily implement mobile versions for my application.

Media content stored in the deployment folder This solution is neither secure, scalable or efficient. Another solution is critical.

Complex database handling The application requires a lot of complex queries. The more advanced the business logic gets, the higher gets the demand for a simple and intuitive way of "crudding". Also, performance and scalability is essential. The faster and more scalable the better. Maybe its time to reevaluate the old relational database and compare it with a no-sql approach.

Comparing solutions

To solve these problems I need to rethink architecture 2.0 completely. One attempt to help make the front-end code better is to move more behavior from the back-end to the front-end, creating a thick client. A thick front-end client can be built using a javascript framework that supports the MVC design pattern. This pattern has showed its elegans since it was introduced in 1979 by Trygve Reenskaug. The problem with the http responses being html pages encourages me to introduce a RESTfull api on the server side, who's responsibility is to return json or xml objects, given some url. This could solve a lot of problems, because now, suddenly I would have a public API that can be used by 3rd party clients, and my mobile applications. In addition, the high bandwidth waste would be minimized, because the amount of data sent in responds is highly reduced. However, it does require that the javascript MVC client knows how to build the web pages given the data offered by the REST api. As for the media content, it should be stored in the cloud by some cloud storage service like Amazon or Windows Azure. Finally, the data tier needs to be experimented with. I should try some various technologies and implementations to verify what is the best solution for persisting Shredhub.

Chapter 6

Architecture 2.0

In this section we will discuss the design and implementation of Architecture 2.0.

An imprint decision is when and where to put script loading tags. Loading scripts blocks the page from loading other recourses and rendering. THere are many options: (this is old though=) http://www.stevesouders.com/blog/2009/04/27/loading-scripts-without-blocking/.

Performance enhancement: minifying and compressing html,css and js. GZIP compresses shit by identifying similar strings. the more matching strings found, the smaller the file can be compressed to.

Cold have chosen to fetch all the HTML at once. This can be done in two ways: Fetch each HTML template one at a time, or merge all together (can be done when deploying) and fetch the whole thing then. However, I chose to lazily fetch them and cache them in the browser (guess require does this?).

Chapter 7

Performance testing

The major goal of an interactive web application is to minimize the time a user has to wait for a response when an action is initiated. An action might be that the user clicks a button, moves a slider, presser the enter key in a search field etc. This is called the response time (often referred to as round-trip time), and is measured by the time it takes from when the user initiates the action, until the result is completely visible in the browser. The response time is a significant factor that affects the users experience of interacting with the web application.

The response time depends on many performance factors in the particular web application. The most important ones being:

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

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

Client-tier efficiency concerns how fast the browser can render a web page and request execute the given javascript instructions

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

The server's performance is also dependent 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.

7.1   Hardware and software used for testing

Bla bla

7.2 Test cases

To be able to get a proper overview of the various pros and cons of the two web-app architectures, there has to be done many independent test cases that are meant to investigate various performance properties of the system.

7.2.1 Test 1 - Page loading

Goal: To determine how fast a particular web page in the web-app is fetched from the server and rendered in the browser
Tests: Response time/round-trip time
The page loading test is meant to investigate for how long the user has to wait from the time he enters a url in the web browser, until the whole page is completely rendered in the browser. This test is highly important, because page-loading time is one of the most fundamental properties of a web application. Users most often don't have patience to sit around and wait for the web page to load, which in some cases could result in users abandoning the site in favor of other web-app competitors.

The page loading test is performed actively by a human user. In this process, the tester will write the url in the address bar of a web browser, press enter and wait for the whole page to completely load. I have chosen to use the Chrome web browser to perform the testing, because it comes with an excellent performance tool, namely the Chrome developer tools [**chrome**]. This tool not only calculates the complete response time, but also the time spent for each individual web resource that is fetched from the server. Both Mozilla Firefox and Internet Explorer has 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.

The urls that has been tested are:

1. *www.shredhub.com/*

2. *www.shredhub.com/theshredpool*

3. *www.shredhub.com/shredders*

4. *www.shredhub.com/shredder/1234*

5. *www.shredhub.com/shredder/1*

These are the five main pages of the web application and requires server interaction for both of the web-app architectures. 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 I perform the page loading tests. Note the final url in the list above is almost the same as the one on top of it. The difference is that the last one is referring to the "profile" page for the currently logged in user. This is done to trigger some interesting results that might occur because most the data that is fetched for this url request comes from data cached in the session object.

The first test is the initial loading of *www.shredhub.com*. This page consists of a login button, some images, formatted text and thumbnail pictures of the top 9 most popular shred-videos. The page has a few interactive functions like picture animations and functions to open up and hide a new window to display a video when a thumbnail image of a shred is clicked.

Table 7.1: default

	Architecture 1.0	Architecture 2.0
Fetching markup and scripts	260ms	420ms
Fetching images	81ms	150ms
Total responds time	405ms	570ms
Time spent on server	63ms	55ms

Architecture 1.0 performs best in this test. The reason is that architecture 2.0 loads a whole lot of small javascript files as well as html template files that are meant to be stitched together in the browser. Each such request requires an explicit TCP connection with the server. Therefore, architecture 2.0 spends some time pulling files from the server, while architecture 1.0 spends less time on this, because it needs less data files. However, less time is spent on the server for architecture 2.0, because in this case the server doesn't have to spend time rendering a complete html page. This might lead to better backend scalability.

7.2.2   Test 2 - Interactivity

Goal: Determine the response time for various user-actions offered by the web-app
Tests: Response time/round-trip time
This test is investigating the responds time spent when a user actively performes a specific task on a given web page. Unlike test 1, which is investigating load time when complete web pages are requested, this test is only concerning minor interaction actions, possibly not resulting in a complete page request from the server. The actions that are tested are:

1. *The user hits the log in button*

2. *The user adds a shred*

3. *The user opens up a shred*

4. *The user comments a shred*

5. *The user rates a shred*

6. *The user deletes a comment on one of his shreds*

Like in Test 1, I have used the Chrome development tool to investigate the round-trip time for each user action.

7.2.3 Test 3 - Database performance and scalability

Goal: Determine how fast a given database query executes, and how many simultaneous queries can be executed
Tests: Database throughput, efficiency and scalability

This test investigates the pros and cons for using a key-value based database versus a relational database for the application. Architecture 2.0 is built with MongoDB [43] which is a document-oriented key value database, while Architecture 1.0 is built with PostgreSQL[40] which is a relational database management system. All the queries that has been executed in this test has been done with various numbers of simultaneous executions. This is to simulate parallel database execution so that the database implementation's scalability characteristic can be determined. The testing is done by using Apache JMeter, which is a tool to perform performance testing on apache server-based applications.

The queries that has been used for testing are:

1. *Read - Get a shredder based on id*

2. *Read - Get all fanees for a shredder based on id*

3. *Read - Get suggested fanees for a shredder based on id*

4. *Write - Add a new shred*

5. *Write - Add a new fanee relationship*

6. *Update - Increase shredder level for a shredder*

7. *Update - Increase rating for a shred*

8. *Delete - Delete a comment on a shred*

All of the queries has ben perfumed with the following amount of simultaneous executions: *1-5-10-100-1000-10000-50000-100000*.

7.2.4 Test 4 - Server scalability

Goal: Determine how many concurrent users the server can support
Tests: Scalability of the backend system

This test is performed by creating dummy requests that executes a sequence of actions on the web page in random order. The actions are meant to simulate normal flows of user actions, to get a best-as-possible view of how well the server scales. Again, the tests are created and monitored by Apache JMeter. The following user actions has been performed:

1. *The user logs in*

2. *The user uploads a shred*

3. *The user watches a shred*

4. *The user comments a shred*

5. *The user accesses the url www.shredhub.com/shredders*

6. *The user clicks on a particular shredder*

The test is performed by a various amount of simultaneous executions: *1-5-10-100-1000-10000-50000-100000.*

7.2.5 Test 5 - System profiling

Determine the throughput and efficiency in the various parts of the system to identity possible bottlenecks

7.3 Results

Part III

Conclusion

Chapter 8

Conclusion

In this thesis I have been comparing various software architectures for web applications that has a rich and interactive web interface and that has high demands for scalability and responsivity. The application has many of the requirements that is normally seen in a modern web application, like interactivity and user participation. Also, the application should be able to offer its functionality as a software-as-a-service.

The goal for the project was to find a software architecture that would best succeed in delivering these requirements. Therefore, I chose to start out with designing a classical three-layered architecture that has been regarded as the best-practice web architecture for the last decades in the web community. However, seen that the classical way for building web apps has seen a paradigm shift with the rise of "web 2.0" I wanted to compare this design with a more modern thick-client web architecture. Both designs where built with the same goal, and they where both optimized in their separate, best suited way. Finally a comparison job was introduced to find the pros and cons for each design.

When comparing the designs, performance was the common goal. The architecture had to deliver its service as efficient as possible. I verified the scalability behavior when more and more requests where fired at the server. The responsivity was tested by actively requesting various resources and measuring the response time. Good results where given by using appropriate testing tools that created dummy requests and flooded the application with requests, both directly to the backend and also through the user interface. This way I got separate tests for the database, the server api implementation, as well as the front-end implementation. Finally the I wanted to see how well the two architectures worked when they where deployed in a cloud environment. The cloud service distributed the application to multiple servers, which has a big impact on how the application performs. The same tests where run on both applications and the results where compared. A final goal with the designs was targeted against the source code itself. Not only was it important that the application performed well, but the software architecture had to be maintainable. Therefore, I compared how the various source codes turned out to be in terms of, flexibility and complexity.

The result was that architecture 1.0 had a very efficient database implementation when run locally, but the front-end became a bottleneck due to the high amount of data returned for each request. This was expected, because the server always responded with a complete HTML page. When introducing a noSQL database for architecture 1.0, I got lower efficiency. However when I deployed architecture 1.0 in the cloud, the system worked better with the noSQL database. It could tolerate many more requests then the postgreSQL implementation. This is probably because cloud environments are optimized to perform well with noSQL databases.

Architecture 2.0 did not achieve the same database performance when run locally as architecture 1.0. However, the front-end was much more responsive. This was clearly because the amount of data sent back from the server where much less then with architecture 1.0. Also, the amounts of http requests where reduced with architecture 2.0, which was a result of the javascript MVC front-end architecture. When architecture 2.0 was deployed in the cloud, it got better performance results then with architecture 1.0, both in terms of scalability and response time for independent requests. I will conclude with stating the architecture 1.0 performed better then architecture 2.0 when deployed locally, and with a small number of simultaneous users. Architecture 2.0 performed best when deployed in the cloud, and achieved higher performance in both scenarios, when the number of users where many.

When it came to code flexibility and complexity, architecture 2.0 had a more compact design, because most of the source code lies in the frond-end. Even so, the code structure was nice and decoupled with intuitive responsibilities and object abstractions. This was a clear improvement from application 1.0, because in this design the front-end code was very complex and lacked code structure. Design patterns weren't even an option in the latter approach, because the way the javascript code used was only as separate functions with no object reuse. In architecture 2.0 however, I could often find places where design patterns had a natural fit. The only pitfall I can see with architecture 2.0 is that the source code is for many developers who are not very familiar with javascript difficult to understand, since the language (which is referred to as a functional-oriented language) is very different from an object-oriented language like java or c++. I will conclude by stating the architecture 2.0 had a flexible and less complex code structure then architecture 1.0, because the application required many dynamic and interactive GUI components. This is better to implement with a thick-client javascript MVC architecture, then with a java-based thin-client architecture with a template language like JSP. Also, noSQL databases works very good with a javascript architecture, because of syntax similarities.

When it comes to being able to deliver application service to third party clients and mobile applications, architecture 2.0 already had this enabled through a restful server API. Architecture 1.0 did not have this, so to enable it, I had to implement separate controllers on the web tier that would check each request to see if they where not from a web user, the server would respond with the appropriate data-format (like JSON or XML). This

resulted in lots of controller handlers, but a flexible solution if one would require support for both html page responses and xml/jSON responses.

Chapter 9

Future work

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).