MASTER THESIS

Extending web frameworks to support HTML5 technologies – The case of JavaServer Faces

Master Software Engineering

at University of Amsterdam

Author:
Mark van der Tol

July 25, 2012

Thesis supervisor: Dr. Tijs van der Storm
Internship supervisors: Arjan Tijms MSc
Klaas Joosten
Availability: Public

Summary

New technologies as HTML5 are getting more and more common. Developers starting new projects want to make use of web frameworks and companies often want to stay with the web frameworks they are using today since they have invested a lot of time and money in it, but still want to have the possibility to make use of new HTML5 techniques in their current applications.

From this situation the research question arose: How can HTML5 technologies be used inside applications written using server-side web frameworks? Changing to another framework that is specifically designed for rich web applications will require a large investment and has other disadvantages, therefore continuing with these existing frameworks is desirable. It will allow to slowly adopting these new technologies into new pages, while the old pages still keep working without.

The web has evolved from a network that was designed to make it easy to share information to a network that is used for complete web applications. From the beginning, the user only had to install a web browser to be able to access the web. The user experience in desktop applications kept evolving, but the user experience in web applications has stayed behind for a long time. Bridging the gap, web applications increasingly resemble desktop applications.

Incompatibilities between the browsers and the slow evolution of them have been a large issue for web developers. Because of that the popularity of browser plug-ins such as Flash grew. To bring these functionalities back to the browser W3C is working on the HTML5 standard.

To limit the scope of the research only one web framework, JavaServer Faces (JSF), was used for the experiments. With that framework three HTML5 features, websocket, web storage and canvas, have been researched. Proofs of concept have been made for the three HTML5 features in JSF. For each feature hypotheses have been made. One of the hypotheses for every selected functionality was that it was possible to make use of that functionality without making changes to the JSF framework.

This thesis shows that it is possible to integrate these three functionalities into a web application written in JSF. For certain functionalities it was required to find a workaround to get it working. Therefore, there is room for improvement in the JSF framework. Companies that want to start making use of HTML5 inside JSF can start to make use of that functionality. Extrapolating all the results to other frameworks hasn't been possible, since the large differences between the different frameworks. The research has shown that it is possible to make use of HTML5 technologies inside server-side frameworks.

Preface

With this thesis, my Master Software Engineering at the University of Amsterdam ends. I wrote this thesis in a period five of months. During the first three and half months, I spent three days per week on the thesis project. During the final one and a half month full time was worked on the thesis project.

I want to thank Zanox-M4N for allowing me to work on my thesis at their organization, especially because my research wasn't related to their daily operations. I want to thank Arjan Tijms and Klaas Joosten for the lengthy discussion we had about the project and all the feedback he has given. I also want to thank the other members of the development team for their feedback, support and for the nice time I had there during the period I was there.

I want to thank my supervisor at the University of Amsterdam, Tijs van der Storm, for the guidance and feedback he has given during the project. I also want to thank the teaching staff from the university for the courses they have given.

Mark van der Tol

Table of Contents

1 Introduction

The web is used for so many things. From putting online your personal homepage, to having a large web application for a large business. All are built from the same building blocks. Frameworks have come in to aid the developer developing for a specific need.

The web is continuously evolving. Much more things can be done in the browser today than could be done a couple of years back. The possibilities of the browsers keep extending. To leverage these new functionalities, frameworks have to be extended. Because these features add more functionality to the client-side, it is hard to integrate these features into frameworks that focus on server-side development.

During this research is looked at how the new client-side functionalities can be used within an existing server-side framework. There are companies who have invested heavily in their applications which are making use of server-side web frameworks. Therefore, it is important to get an idea how to keep these frameworks relevant in a period where the web becomes more interactive. This allows applications to be updated to make use of rich interactions, instead of being rewritten. This is needed for existing applications and for new applications.

This project considers the server-side framework JavaServer Faces (JSF). This framework is used within the organization the thesis project is performed. By making proofs of concept it is shown whether it was possible to integrate HTML5 functionalities into JSF. This is done by letting the proofs of concept prove the hypotheses.

The thesis starts with the motivation. The motivation describes what the current and the desired situation is, and what the research question is this thesis will answer. It also describes the contribution the results of this thesis will give to the field. After that, the background for this research is described. The background consists of the background of the web, how the web was changed to accommodate the richer level of interaction, background behind JSF and the differences between client and server-side logic. After that is described how the research is performed. Then there are the design decisions given for the proofs of concept. In the final chapter are the answers to the hypotheses given and the chapter contains an analysis and conclusion. It also contains directions for future research.

2 Motivation

In this chapter, there is first an overview about the current situation. That is followed by the desired situation that is currently still out of reach. After that is the main research question. Finally, there is the contribution this research will give.

2.1 Current situation

Many different web frameworks are used to develop web applications. These web frameworks are being used for many years and large applications have been built on top of those applications. Currently new techniques are getting more commonly used inside web applications. Many of these new techniques were standardized together inside the HTML5 specification.

JSF is such a framework. JSF was originally designed to be a server-side GUI based framework.[JAVA COMMUNITY PROCESS, 2004] The developer could create a page composed from components and allow the view logic to interact with these components. JSF provides a library of common components. These components include components for basic HTML elements such as form input elements, and somewhat more advanced components such as a component that can render a table.

To improve the interactivity AJAX support has been added to JSF. It allows performing a lifecycle on the server for a selection of the components on the page and then returning the new HTML for the updated components. This allows sending information to the server and allows the server responding to it, without loading a new page.

One example of client-side functionality that is already possible is form validation. This is possible to do with the build-in AJAX functionality of JSF. When the user fills out a field inside a form, it is directly verified at the server by sending an AJAX request. The only thing the developer has to do is to add a client behavior to the definition. The field can then be highlighted to show that the value won't be accepted when the form will be submitted. However, other client-side functionalities haven't found their way into the specification and therefore still require research.

2.2 Desired situation

AJAX is one way to implement rich interactions. However, as explained in previous paragraphs, richer interactions are becoming more commonplace at the moment. Examples are client-side APIs such as web storage and canvas elements. These interactions move logic that classically would be done at the server to the client. Because these APIs run completely at the client, they cannot be integrated into the request lifecycle. This is because the lifecycle runs completely at the server. It is also unwanted to communicate with the server for these client-side APIs, because that would introduce latency for the user and increase the server load. Therefore, the components should be able to run stand-alone in the browser.

The websocket API for example changes the way the application in the browser communicates with the web server. Instead of using an HTTP request for every interaction, a single connection is

opened to the server to send interactions. A websocket connection is full-duplex and can be used stateful. In contrast a normal HTTP connection isn't full-duplex and stateful. During a normal HTTP request the client sends a request and then the server will send its response. If the server has updated information, it cannot be send to a browser until that browser places a request.

2.3 Research question

The main question this research tried to answer was: **How can HTML5 technologies be used inside applications written using server-side web frameworks?** This question arose from the problem described under the section "Desired situation".

2.4 Contribution

Web frameworks are commonly used when web applications are developed. Because of the popularity of these frameworks there can be concluded that this is a convenient way for developers to develop web applications. Having the possibility to make use of HTML5 inside such a framework will keep these frameworks relevant.

Another reason to research this subject is because many companies have invested a lot of time and money in the web applications they are currently using. Therefore many companies wish to continue using these existing frameworks. Changing to another framework that is specifically designed for rich web applications will require a large investment. Making it possible to migrate to HTML5 technologies slowly is desirable. Having the possibility to only have the new pages use these new features, while still keeping the old pages functioning without these new technologies is a common desire.

To make it possible that these new technologies are introduced into existing frameworks, research is required into these frameworks. To allow the research to be deep enough, instead of too wide, only a single web framework is considered. This web framework, JSF, is a very commonly used framework and therefore the results will be useful for many people.

3 Background

The first part of this chapter describes how the web started and evolved into an application platform. After that are the differences between doing interaction and business logic at the server-side and doing the same logic at the client-side. Finally, there's a description what JavaServer Faces is, how it evolved to the current version and there's a high-level overview of the architecture.

3.1 Web

First is described what the web was developed for, thus a background is created for what server-side web frameworks were created for.

3.1.1 For what goal has the web been designed?

The World Wide Web was designed to make a shared information space through which people and machines could communicate.[FIELDING and TAYLOR, 2002] It was quickly identified that it was important to allow users to work with information as they do normally.[BERNERS-LEE, 1996] Forcing users to deal with information in the same way computers deal with information wouldn't have become successful. It was important to have a low barrier for users to get quick adoption.[FIELDING and TAYLOR, 2002]

The intended users of the web were located around the world at universities and physics labs. The machines differentiated from workstations to supercomputers. There were a lot of incompatibilities between the data formats used for communication. The goal of the web was to bridge this gap and allow every computer to connect with the network.[BERNERS-LEE, 1996]

A choice was made to decentralize the web and use hyperlinks to other documents. The choice for using hyperlinks offered the required flexibility. It allows pointing to all kind of documents. At the time it was common to have centralized link databases. These centralized link databases have the advantage that links placed in other documents can directly be maintained, but have the disadvantages that it will not scale. Because the web is designed to be decentralized it is possible that targets of links get unavailable.[BERNERS-LEE, 1996][FIELDING and TAYLOR, 2002]

The only thing the user has to have to access the web is an internet connection and the only thing the user has to install is a web browser. Because browsers have become available for all platforms it made the web even more popular. By allowing multiple platforms it has been kept in mind that forcing users to a specific platform wouldn't work.[BERNERS-LEE, 1996]

Making it easy for the web to evolve was an important goal. To describe the functionality of the web multiple standards were made. Examples of these standards are HTTP and HTML. These standards are limited to their own scope. This allows the standards to evolve independently. Parts of the design can be replaced without the need of updating the other parts.[BERNERS-LEE, 1996]

3.1.2 What is the web used for today?

The paper "web browser as an application platform" [TAIVALSAARI *et al.*, 2008] identifies three phases in which a web page can be. In the first phase, the web pages are more like documents. These pages only contain text with some static images, without any interactive content. In the second phase the web pages have become more interactive. There is in these pages some use of animated graphics, plug-in components or JavaScript. Web pages in the third phase start to resemble desktop applications. These kinds of applications can be called Rich Internet Applications. These often limit the amount of link-based navigation and manipulate the contents of the page instead.

Because the web became more interactive, commercial interest in using the web increased. Advertisements or selling services could now be done over the web [TAIVALSAARI *et al.*, 2008] (as for example Zanox-M4N does). Especially the increased interactivity and graphics capabilities were drivers for this phenomenon.

This distinction in three phases allows easy distinction between the various levels of interactivity. It shows roughly how the web has developed. Users today expect from a web page an increasingly higher level of interactivity.[FARRELL and NEZLEK, 2007]

Most web pages of today are in the second phase, although an increasing number of pages are made that fit in the third phase. The modern browsers, even the browsers on mobile devices, offer more functionality. Because of the extended functionality more web pages are written as Rich Internet Applications and more applications are made for the web that before would have been made for the desktop.

The user experience in desktop applications kept evolving, but the user experience in web applications has stayed behind for a long time. To bridge the gap between the user experience of desktop applications and the user experience of web applications, web applications start resembling desktop applications.[FARRELL and NEZLEK, 2007]

Another trend in web development is Web 2.0. Web 2.0 are technologies that allow easier collaboration and sharing on via the web and are as responsive to user requests as desktop applications.[ANKOLEKAR et al., 2007][JAZAYERI, 2007] Web 2.0 websites put the user central. There isn't a distinction between the producers of the information and the consumers of the information.[JAZAYERI, 2007] Users submit the content for the web page. Other users use that information placed by others. Each contributor gains more from the system than he puts into it.[ANKOLEKAR et al., 2007]

3.1.3 What standardization effort has been done?

A big issue when developing for the web is the incompatibilities between the different web browsers. Multiple vendors are making browsers, whereby most vendors support multiple versions of the browser.[TAIVALSAARI *et al.*, 2008] These incompatibilities can be caused by disregard for official standards, because standards are misinterpreted or too ambiguous, or because there are no standards that offer certain important functionality. In the latter case it often becomes standardized when other browser vendors also see the importance.[TAIVALSAARI *et al.*, 2008] Until the functionality is standardized each browser will have their own implementation.

Scattering of the implementations was seen as threat. Therefore, committees are standardizing the technologies used for the web. Today the organization named W3C does most standardization of web technologies. These committees consist of employees of companies that are using these

technologies. Now they also standardize related technologies, such as Document Object Model (DOM) and XMLHTTPRequest.

To make browsing the web secure, multiple standards have been defined. These standards include cross origin policy and limited access to the local system of the user.[TAIVALSAARI *et al.*, 2008] It is strongly advised to validate all the data sent from the client to server on the server. Client-side security can easily be bypassed.[PAUL, 2007]

To allow interaction with the document JavaScript support has been added to all major browsers.[TAIVALSAARI *et al.*, 2008] Syntactically JavaScript resembles the C and Java programming languages. However, in practice JavaScript is a much more dynamic, interpreted language that has features from other highly dynamic languages such as Smalltalk and Lisp.[MIKKONEN and TAIVALSAARI, 2008] JavaScript is formalized under the name ECMAScript. The standardization of JavaScript isn't done by W3C, but is done by ECMA.[ECMA, 2011]

3.2 How has the web changed to accommodate a richer level of interaction?

To accommodate richer web applications the feature set of the browsers has been extended. Many of these features are standardized in HTML5. In addition, many plug-ins have been made to extend the functionality of the web browsers.

Many vendors have made technologies that extend the functionalities of the browser via a plug-in. Examples of these technologies are Adobe Flash and Microsoft Silverlight. These plug-ins offer functionalities that the browser doesn't offer. An example of such functionality is extended multimedia support. Although these plug-in have increased the capabilities, they have decreased openness.[VAUGHAN-NICHOLS, 2010]

To bring back the openness of the web W3C has started with the HTML5 standard. It includes an API for video, audio and vector graphics. Although HTML5 is still a markup language for web pages, the support for web applications is a major goal.[VAUGHAN-NICHOLS, 2010]

One disadvantage of the client-server model is that the client has to load a new page on every interaction. This reduces the interactivity.[FARRELL and NEZLEK, 2007][MESBAH and VAN DEURSEN, 2007] Another disadvantage is that the web is stateless. Both these properties result in a lot of redundant data being passed between the client and the server, wait times after each interaction and that pages have a start and stop feel due to their multi-page interfaces.[FARRELL and NEZLEK, 2007] By using modern technologies such as XMLHTTPRequest the web page behaves more like a desktop application.[FARRELL and NEZLEK, 2007] [TAIVALSAARI *et al.*, 2008] These techniques have been coined AJAX. AJAX is an abbreviation for Asynchronous JavaScript and XML, although it can be used synchronous and other technologies instead of XML can be used. The term AJAX has superseded the term dynamic HTML (DHTML).[MESBAH and VAN DEURSEN, 2007]

It is possible to move some of the user interface logic to the browser, it is possible to move almost all user interface logic to the client and it is possible to do something in between. Moving all the logic to the browser is hard. Adding some small AJAX functionality to a web page to augment the experience is much easier. Therefore, the latter is becoming more common nowadays.

The performance of JavaScript was considered to be a problem for developing web applications.[TAIVALSAARI *et al.*, 2008] Many browser vendors started to improve their JavaScript performance to make the use of JavaScript suitable for more tasks. Nowadays the JavaScript performance is significantly improved. This enables the use of JavaScript for making rich web applications.

Because browsers weren't originally meant to be used as an application platform some problems could arise. An example of an encountered problem is that when using a web browser to show an application, is the back and forward button. The back and forward button can feel unnatural when they are used in a web application.[TAIVALSAARI *et al.*, 2008] For example when the user has placed an order, clicking on the back button doesn't cancel the just placed order. This is resolved by a feature that is introduced to make it possible to interface with the browser history from JavaScript.[WHATWG, 2012]

3.2.1 Limitations of the current web

A very large problem in web development is the incompatibility between the browser implementations. These incompatibilities are in various areas. Most of the functionality of the web is standardized, but not all of the functionality. Also not all standards are implemented or are implemented correctly. This makes writing code that runs well in different browsers difficult.[TAIVALSAARI *et al.*, 2008] A problem with standardization is that the standardization process is slow. This is because it will standardize proven existing implementations, not invent completely new ones. The planning of HTML5 states that it will be at least a decade later before HTML5 is formally standardized.[VAUGHAN-NICHOLS, 2010] This of course doesn't mean browsers can start using HTML5 technologies, but it means the standard is still evolving.

Libraries have been made to deal with the incompatibilities between the different browser implementations. Well-known examples are jQuery and Prototype. Libraries like these make an abstraction over the browser interface to deal with the incompatibilities.

Many commonly required features still aren't available in the most popular browsers. Such features are streaming video, access to the local files or using hardware of the computer such as a webcam.[TAIVALSAARI *et al.*, 2008] Plug-ins provide these features now.

These plug-ins have to be installed before they can be used. A problem is that plug-ins are often not available for all platforms. Another problem is that plug-ins aren't standardized. The creator of the plug-in decides what features it will have and makes the release schedule. Because these standards aren't open for anyone to use, users of the plug-in are dependent on the plug-in vendor for updates to let the plug-in evolve.[JOBS, 2010] Plug-ins are also the most common source of security leaks in web browsers.

Because plug-ins aren't open, aren't portable to all platforms and because they are a common source of security leaks, the HTML5 standard tries to remove the need for plug-ins. This hasn't succeeded fully yet. Some browsers even have removed the possibility to have plug-ins. One of these browsers that doesn't support plug-ins is the browser included on the iPhone.[JOBS, 2010]

Maintainability of web applications is a big problem too. The application logic is mixed up with user interface components. This makes it harder to see the control flow and makes the maintainability lower. Also information hiding is hard to do when writing scripts in JavaScript that interface with the DOM. It is hard because the DOM is completely available to all scripts. The JavaScript language

lacks features needed for information hiding, such as making variables protected or private. Support for categorizing the code in packages or namespaces is also unavailable. When writing scripts it is also hard to separate the different technologies. It is often required to make use of HTML or XML inside JavaScript.[MIKKONEN and TAIVALSAARI, 2008]

3.2.2 Rich Internet Applications

Because the web has evolved, companies have identified the possibility to make complete applications that run on the web. These kinds of applications are called Rich Internet Applications. These pages aren't designed as documents, but resemble rich desktop applications. Many interactions by the user no longer require a reload of the page.[FARRELL and NEZLEK, 2007]

Some companies have designed their pages using plug-ins to make Rich Internet Applications. The most popular plug-ins are Flash and Silverlight. Because of HTML5, it is often no longer needed to use plug-ins to achieve this. The compatibility between the browsers is improved, because no plug-ins have to be used.

Well-known examples of web applications that run in the browser are GMail and Google Docs. GMail from Google is an online email client completely implemented in the browser. It competes with other online email services and desktop applications such as Microsoft Outlook. When the user clicks on an email, the page doesn't reload. Outlook only retrieves the email from the server. The other example is Google Docs. It allows the creation of documents within the browser. It competes with the desktop suite Microsoft Office. Because it is on-line the documents are available everywhere and it allows documents to be shared easily.

The advantages of web applications above desktop applications are that the end user doesn't have to install the applications, it is relatively easy to make the application compatible with all platforms, it is easier to deploy because no installation is needed and it is available on all devices with an internet connection. The disadvantages are that less access is available to the hardware, a lot of effort is needed to make the application run properly in all popular browsers[FARRELL and NEZLEK, 2007] and end users are forced to upgrade to the latest version.

3.3 Differences between browser- and server-side logic

There are large differences when business and interaction logic are placed on the server than when logic that originally was on the server is moved to the client. These differences show themselves for the user, because when logic is performed at the client-side there is a different user experience. But also for the development of the application there are large differences.

First is looked at what the differences are for the users in terms of user experience. After that is looked into what moving logic from the server to the client means for the development of web applications.

3.3.1 User experience

An important factor to keep in mind when designing web applications, especially when deciding where to perform the logic, is the application performance. In classic web applications almost everything is done at the server, but today it is possible to move some logic to the client. This will reduce the time an interaction will take and will reduce the load on the network.

A very important metric for user experience, when considering whether AJAX will or won't be used, is time required for an interaction. Improving the interaction time will increase the satisfaction of the end users. For what an acceptable interaction time is, no consensus is reached. A study on tolerable waiting time: how long are Web users willing to wait?" [NAH, 2004] argues that acceptable waiting times lay between 2 seconds and 20 seconds. The author thinks 2 seconds is an acceptable time for normal interactions. It argues that 100 millisecond is about the time the user feels that it is reacting instantaneously. 1 second is the limit where the user's flow of thought isn't interrupted. When an interaction takes more than 10 seconds, then users lose their attention.[NIELSEN, 1993] Giving feedback for operations that take longer than 1 second, for example by showing a progress bar, makes the user more willing to wait.

Research from Google has shown that when interactions are faster (even differences of 200 milliseconds makes a difference) more interactions are made by the user.[BRUTLAG, 2009] This shows that even small performance improvements can have an effect on how users make use of the system.

When AJAX is used, less data has to be sent from the server to the browser and the performance of the network can also improve. The server only needs to send data for the parts of the page that have been changed, instead of a complete page. Parts that haven't changed don't have to be sent again. Examples of parts of the page that won't change in most cases are the header of the page and the sidebar of the page. How big the advantage is depends on the size of the static part and how often the user performs an interaction.

In many cases it is required to include the logic that is required to perform these rich interactions. This logic is often included as a separate JavaScript file. This will increase the load time of the initial page load, even before the user has made an interaction. It has to be determined whether this weighs up against faster interactions for the user.

Because some operations no longer require server-side logic or the server doesn't have to return the parts of the page that haven't changed, the server has to deal with less load. Simpler or less servers can then be used to serve the web application. The aspect about the load is hard to measure, since it depends on many variables.

3.3.2 Development

In the classic way of developing web applications, the server generates a page at the server and then sends it to the browser. Once the user performs an interaction on the page, for example when he submits a form or clicks a link, then a completely new page is loaded. On the web servers languages such as Java, PHP and C# are commonly used to generate the web pages.

AJAX is one of the functionalities that can be used to improve the interactivity of a web page. AJAX is a feature that makes it possible to send HTTP requests from JavaScript. [VAN KESTEREN, 2012] It is possible for the script to include data in the request, such as the values the user entered in a form on the page. The server handles that request and sends a response back. JavaScript can then handle that response. XML or JSON are common formats used to format the response, but other formats are also possible. The script in the browser is then able via a callback to perform operations.

A problem with using AJAX is that there is another interface with the browser and the server. The interface has to be designed, along with the logic that has to be executed at both sides. This will require more development effort and increases the probability of bugs.

To make the development of a web application more efficient frameworks were developed. An example of such a framework for Java is JavaServer Faces. [BURNS and SCHALK, 2010] A common feature of frameworks is to make it possible to separate the business logic from the view logic. This makes it easier to design code that facilitates reuse.

To increase the interactivity of web applications JavaScript could be used. When JavaScript is used, it is possible to integrate it directly within the document or to reference it as a separate file. When scripts are used in multiple pages, it is advisable to put them in a separate file to make the reuse easier. To separate the view from the business logic it is also advisable to separate the scripts from the HTML by putting the JavaScript in a separate file.

A problem that makes JavaScript hard to use and code written in JavaScript difficult to maintain is that JavaScript is weakly typed. It is hard to validate whether referenced objects exist before execution.[MIKKONEN and TAIVALSAARI, 2008] Static languages such as Java can validate the types of the objects at compile time, instead of runtime.

As stated earlier, another big problem that developers have to deal with when developing web applications is that browsers have lot of incompatibilities.[MIKKONEN and TAIVALSAARI, 2008] These incompatibilities are mostly in the DOM interface, not in the JavaScript language itself. To increase the efficiency of development, the browser incompatibilities are hidden behind abstractions. This makes it possible to use the functionality with a consistent interface, while there isn't the need to deal with the incompatibilities every time they are required.

One way to speed up interactions is to move the logic from the server to the browser. Then the user doesn't have to wait for the server to complete a response. The developer has to be aware that it is sometimes impossible to move the complete functionality from the server to the browser. Examples of cases where this isn't possible are when data needs to be validated, data has to be persisted in a database or when data or operations are required that are only available on the server. As stated earlier, data validation could be bypassed on the browser by disabling JavaScript. To ensure the consistency of the data, it is always needed to validate user entered data at the server.[PAUL, 2007] A possibility is to perform such validations both on the server and the browser. To make still it possible to provide interactivity with the server for these features that aren't possible in the browser without loading a new page, AJAX techniques can be used.

3.4 JavaServer Faces

The Java Community Process maintains the JSF specification. The specification prescribes a framework that can be used to develop user interfaces for web applications. The maintainers of the specification are developers from Oracle and other companies using JSF.

JSF is part of Java Enterprise Edition (Java EE). This means it will integrate well with other Java EE technologies, such as the Java Persistence API (JPA) and Enterprise JavaBeans. This also means that JSF ships with every Java EE server by default.

3.4.1 Features

The main feature from JavaServer Faces is the view based on MVC and components. The files that declare the view are in a special template language called the Facelets view declaration language, which allows binding attributes of components with objects from the model. The controller dispatches the requests directly to the appropriate view. JSF provides the controller. The model

contains the business logic and non-user interface code. The view contains the pages in a template language.

The model that is bound to the view can make use of dependency injection frameworks. The model resolution can be plugged by multiple implementations, including Enterprise JavaBeans and Spring.

An important feature inside the template language is the Unified Expression Language. The developer can bind attributes of components to model objects using this expression language. A separate standard maintains the Unified Expression Language.

JSF introduced the concept of templating to facilitate reuse of view code. It allows defining a part of the page that can be reused in another page. This will allow reusing parts templating code.

3.4.2 History

In 2004 was the first version of JSF released. The proposal for the first version of JSF states that the goals were to: [JAVA COMMUNITY PROCESS, 2004]

1. *Create a standard GUI component framework which can be leveraged by development tools to make it easier for tool users to both create high quality GUIs and manage the GUI's connections to application behavior.*
2. *Define a set of simple lightweight Java base classes for GUI components, component state, and input events. These classes will address GUI lifecycle issues, notably managing a component's persistent state for the lifetime of its page.*
3. *Provide a set of common GUI components, including the standard HTML form input elements. These components will be derived from the simple set of base classes (outlined in #1) that can be used to define new components.*
4. *Provide a JavaBeans model for dispatching events from client-side GUI controls to server-side application behavior.*
5. *Define APIs for input validation, including support for client-side validation.*
6. *Specify a model for internationalization and localization of the GUI.*
7. *Automatic generation of appropriate output for the target client, taking into account all available client configuration data, such as browser version, etc.*
8. *Automatic Generation of output containing required hooks for supporting accessibility, as defined by WAI [Web Accessibility Initiative].*

These goals are still upheld in the current version.[BURNS and SCHALK, 2010]

With the release of 1.2 in 2006, JSF became part of Java EE. Before that, you had to add JSF separately to a project as library. Now it is a default library for each Java EE server.[JAVA COMMUNITY PROCESS, 2008]

In 2009 was the release of the 2.0 version of the specification. The goals for that version of the specification were to improve the ease of development and introduce AJAX functionality.[JAVA COMMUNITY PROCESS, 2010]

By removing the need to use configuration files, the effort required to develop web applications was reduced. What first had to be defined in a configuration file can now be defined using Java annotations. The need to deploy the project again when a JSF file was changed was removed. This reduces the time needed to do iterative development.

The introduction of AJAX functionality was a large feature. It includes changes to the lifecycle in the application to become aware of the AJAX requests. JSF renders the view partially in an AJAX request. JSF sends back to the browser only specific elements of the page, not the complete page. The specification includes the interface for a JavaScript library that deals with the interactions at the browser side.

In 2010, there was the release of version 2.1. This version did not contain large changes. It was just a maintenance release of the specification, fixing small problems in the specification.[JAVA COMMUNITY PROCESS, 2010]

3.4.3 Implementations

Because JSF is only a specification, the implementations are separate from the definition. The specification only defines the public interface and the behavior. Therefore, the implementations can still have completely different internals.

The first implementation is the reference implementation of the specification: Mojarra. Mojarra is an open-source and free to use implementation of JSF. Mojarra is available under the Common Development and Distribution License, and the GNU General Public License. Oracle maintains this implementation.

The second implementation is Apache MyFaces. The Apache Foundation maintains this implementation. Apache MyFaces is available under the Apache License. The Apache MyFaces team claims to distinguish itself from the reference implementation by focusing more on the community and that it does more on innovation.[APACHE MYFACES, 2012]

3.4.4 Architecture

JSF runs in every modern Java Servlet container. Examples of these servlet containers are Apache Tomcat, JBoss AS and GlassFish from Oracle.

The template language that is integrated within JSF, the Facelets view declaration language, is used to define web pages. When the page loads, JSF parses the document at the server-side into a tree. All the phases use this tree during the lifecycle of a request.

An important aspect of the architecture of JSF is the request processing lifecycle. During the request processing lifecycle, a request is split into multiple phases that are executed is a specific order. These are the phases of the lifecycle: [BURNS, E. AND SCHALK, C. 2010]

- Build or restore view tree; at the server-side a tree will be built when it isn't built already. When the request is a post back, then JSF restores the view from the previous request.
- Apply request values from client data; the values sent from the browser are put into the components currently in the tree.
- Validate components; components can have validators attached. The values now in these components are validated. A common validation that developers put on a component is to require entering a value into the component. When the value of one or more of the components isn't valid, it skips the next phases to go directly to the render phase.
- Push values from components to the model; since the values that are hold by the components are validated to be correct, the values can then be sent to the objects bound (using data-binding) to the components.

- Invoke application logic; when extra application logic is required, for example when the user clicks on the submit button of a form in his browser, then this logic is performed in this phase.
- Render the tree; the components are rendered to (X)HTML in this phase and the result is sent to the browser.

3.5 Requirements

When building applications on top of a web framework, there are a couple of requirements that the developer working on that application has to consider. We used these requirements when the proofs of concept were made.

3.5.1 Maintainability

Making changes to the existing framework is undesirable. Therefore, these features should be implemented on top of the JSF framework. Although frameworks are extendable, that doesn't mean that every functionality can be added to it. JSF could be extended by making use of many extension points. It is for example possible to listen for lifecycle events, replace factory objects and add new components that can be used from the framework. When changes have to be made to the framework, then it will require substantially more work to maintain. It makes it harder to upgrade to new versions of the framework, because the changes have to be applied again in the new version. In addition, the changes to the framework have to be tested to prevent the introduction of regression bugs.

3.5.2 Security

While designing software that is available from the internet, security is a very important factor. Web servers can contain information that must not become public. When security measures can be bypassed, then this can have big consequences for the company.

Therefore, it is important that during the implementation a security consideration is made. This means that some operations have to be done at the server and cannot be moved or copied to the browser. Other operations could be done at the browser, but have to be done again at the server. This is because the operations done at the browser cannot be trusted, because the user can tamper with the data.

4 Research method

This chapter explains the performed research. It starts with the methodology that was used. How the results of the research were validated is after that. The features that were researched during the project follow after that. Finally, there are the hypotheses for the selected features.

4.1 Methodology

To answer the question how it is possible to integrate HTML5 functionality into existing web frameworks we made a case study. The case study consists of implementing selected HTML5 features into an existing web framework. There are hypotheses about the design and the proofs of concept test these hypotheses.

The results are both exploratory and confirmatory. It explorers what problems arise when HTML5 functionality is introduced into an existing framework. On the other hand, it tests whether it is possible to integrate these functionalities.

The proofs of concept are built on a single framework. The reason for this is that is that there are large differences between the server-side frameworks. JavaServer Faces is the selected framework. We have selected this framework, because the company where the research is performed, Zanox-M4N, is using that framework for their web applications. This allows getting information directly from developers that are working regularly with that software. For Zanox-M4N, it has the advantage they can make use of the findings for future projects.

The HTML5 features cover a broad spectrum of different scenarios. This is done, because there isn't enough time to cover all features of HTML5. In addition, some features have already found their way into the specification or some third-party framework vendors have already implemented that functionality.

The case study will give qualitative results that will show how to integrate HTML5 functionality into JSF. It shows whether it is possible to integrate these functionalities into JSF and when it is possible, how these functionalities integrate with JSF.

4.2 Validation of results

As described in previous part of this chapter, we give the answers to the questions only by looking at a single framework. Therefore, care is taken to prevent extrapolating results too far. By confirming that what is said is really confirmed by the findings, this is mitigated.

Because in interest of time a limited number of features were selected, the results won't speak for all HTML5 functionality. Also for this feature, care has is taken to prevent extrapolating results too far.

The hypotheses have been written in a way they can be irrefutably confirmed. The proofs of concept will show that it is possible. A proof of concept cannot be used to confirm that something is

impossible to do, only that something is possible to do. To confirm that it is done right, two other developers validate the code.

Because a single person writes the code, the risk exist that the coding qualities of single person are tested. Two experienced JSF developers working for Zanox-M4N reviewed the code to mitigate these risks. These developers have been working for many years with JSF to develop their applications.

By showing the line counts of the source files, it is possible to confirm that the HTML5 functionalities are built on top of JSF, instead of forking part of JSF.

4.3 Selected features

The features that this research looks at are web storage, websocket and canvas. All three functionalities are part of HTML5. These features require more than letting the server send out HTML with some JavaScript. The features will require some logic at the server-side. The kind and amount of logic varies per feature.

We selected these features to give a broad spectrum of HTML5 features. In the interest of time not all HTML5 features could been tested. Websocket changes the way web applications can communicate with the server. Web storage changes the way the data for the user is persisted. Canvas changes how information is presented at the client-side and changes client-side interaction.

4.3.1 Web storage

Web storage is a feature introduced to allow storage of data at the client-side in addition to cookies. Before web storage was available, data for a web application was persisted in a cookie or was stored in a database on the server. This technique adds another way to store data for the client. The disadvantage is that the server cannot access the data directly, because the data has to be explicitly sent to the server. Advantage is that for the storage no connection is required from the browser to the server. There are two kinds of web storage: session storage and local storage.[HICKSON, 2011]

Local storage

Local storage is storage for the current site. The storage is available for all pages from the same site. This includes cases when a site is opened in multiple windows. The data in the storage is persisted, even when the session ends. Cookies also have the ability to store data that is persisted after the session ends, but this information is sent to the server with every request. This makes this method less efficient for large amounts of data.

Session storage

Session storage is storage for the current window. Other pages from the same site can use the storage when a new page is loaded in the same window. When the same site opens in a new window, the browser creates a new storage object for that window. When the session ends, the browser deletes that storage. Besides that the browser sends the cookies with every request, cookies don't allow information to be stored for a specific window either. Cookies can only be used for a specific domain.

Interface

Both kinds of storage have the same interface. The interface allows storing values identified by key. From JavaScript items can be added and removed. In addition, functions exist to show what keys are stored and how many keys are in the storage.

All major web browsers, including Microsoft Internet Explorer 8 and 9, Mozilla Firefox, and Google Chrome have support for the Web Storage interface. Internet Explorer 6 and 7 don't have support for web storage. It is possible to use cookies as fallback in some scenarios, but as mentioned earlier it does have its disadvantages.

4.3.2 Websocket

Websocket is a new functionality that is at the time of writing supported by the most popular browsers, except Microsoft Internet Explorer.[WHEN CAN I USE, 2012] Websocket is a major feature from HTML5 that makes it possible to have a bidirectional connection from the client to the server. It provides a more convenient interface than for example HTTP Push offers, which makes use of normal HTTP requests. Websocket changes the way the browser can communicate with the server.[HICKSON, 2012-4]

The websocket proof of concept shows how we added websocket functionality into a page made with JSF. We offered a fallback for web browsers that don't have support for websocket.

Polling

In the standard model, the browser sends a request to the server and the server responds with an answer to that request. When the information on the page is changed, the page obtains the information in the new state by polling the page in regular intervals. Furthermore, when the user performs an operation on the page a new AJAX request will be required to send this interaction to the server.

Three large disadvantages of this method are that the page only updates after an interval, thus new information doesn't appear directly; when the information hasn't changed there is still polled for changes, thus wasting bandwidth and server load; and when the user performs an interaction, a new connection has to be opened to send that as an AJAX request.

Long Polling

An alternative to polling is long polling. When the page loads, the browser sends an HTTP request to the server. The server doesn't return a response to the request until new information is available from the server. When the request has returned, the browser opens a new request to wait for new data changes.

Long polling solves two of the disadvantages. The first disadvantage is that pages are informed directly when the change occurs, instead of the moment there the page is polls for a change. The second disadvantage is that the scripts don't poll unnecessary when the information hasn't changed. This reduces waste of bandwidth and server load.

The disadvantage of this method is that not all servlet containers (the HTTP server that hosts the application) are capable of dealing with long polling requests. Another disadvantage is that for every user a connection, the server keeps a connection open. This means that the server must be prepared to deal with having a large amount of HTTP connections open.

Websocket

The browser downloads a page that uses websocket just as a normal page. Instead of regular polling for changes, a script on the page opens a continuous connection to the server. When the information changes, the server sends the update directly to the browser over that connection. When there are no changes, the server sends no data to the client.

Using websocket has three major disadvantages. The first disadvantage is that not all browsers support websocket.[WHEN CAN I USE, 2012] At the time of writing the latest released version of Firefox (12.0) and Chrome (19.0.1084) have built-in support for websocket. Internet Explorer 9 (which is at the time of writing the latest released version) and earlier versions don't have support for the websocket protocol. A beta version of Internet Explorer 10 does offer that functionality though. Since Internet Explorer has a large market share, only supporting websocket could be problematic. Therefore, a fallback mechanism would be desirable.

The second disadvantage is that the server software has to have support for websocket. At the time of writing, not all major Java application servers have support for the websocket protocol. GlassFish 3.1.2 does, but JBoss 7.1 does not. Websocket is implemented as an extension to the HTTP specification. The HTTP protocol defines an "Upgrade" header, that allows the client to request for another protocol.[FIELDING and TAYLOR, 2002] Websocket makes use of this header to upgrade an HTTP connection to a websocket connection. The websocket functionality cannot be added by just including a library to the server. The code for handling HTTP request has to be extended to allow websocket to function.

The third disadvantage is that when the browser connects to a server using websocket, the browser keeps that connection open for the duration that the user is on that page. This means that the server must be prepared to deal with having a large amount of websocket connections open. When the web server has a thread for every open connection, then it won't scale to many connected users. Polling doesn't have this problem, because when polling is used every interval, the browser opens a new connection, the browser performs the request and then the connection closes. Therefore, the server has to deal with less concurrent connections.

4.3.3 Canvas

HTML5 introduces a new element with the name "canvas".[HICKSON 2012-3] The canvas element can be inserted into the HTML page just like any other element. The canvas element makes it possible to render graphics directly in HTML pages with the use of JavaScript. Without the canvas element, you would need techniques like Flash to achieve such possibilities.

Canvas interface

The canvas API supports multiple contexts by design. Currently there are two different contexts defined: a 2d context and a 3d context. The 2d context has been specially designed for the canvas element, while the 3d context is based on the OpenGL specification. During the research, we only considered the 2d context, because the 2d context is supported in most browsers, while the 3d context isn't. The 2d context is supported by all browsers that have support for the canvas element, while the 3d context is only supported by Google Chrome and Mozilla Firefox.

The canvas 2d API provides a blank raster that scripts can paint on using a graphics API. [HICKSON 2012-2] Scripts can draw on the canvas various shapes onto the raster. There is also extensive support for rendering fonts. Because the canvas redraws relatively quickly, the canvas element can be used for animations.

The API also provides some means to deal with user interaction within the canvas element. It is possible to query whether a certain point is on a path. This allows the script to respond to clicks on a specific area in the canvas element.

Use

The canvas element can generate richer graphics than was possible before the canvas tag came available. In the standard uses like rendering graphs and images are proposed. Because it is possible to render animations and because it is possible to respond to mouse and keyboard events, it is also possible to make a game that makes use of the canvas element.

When the canvas element isn't available, it is possible to provide a fallback. The browser doesn't show the child elements of the canvas element when the browser supports the canvas element. Thus when the browser doesn't support the canvas element, the child elements become visible and can provide a fallback.

4.4 Design hypotheses

This section lists the hypotheses for this research. These hypotheses describe design assumptions. It assumes these will be possible.

4.4.1 Web storage

Web storage can be used within an application written in JSF to remember values entered in a form.

There are certain forms where the information that the user enters is usually the same as the user entered the last time in that form. Therefore, it would be convenient that the page already fills in that information, so that the user only has to check whether the information is still correct. Web storage could be used to store that information, so that the server doesn't have to store that information and the client doesn't have to be sent the information along within a cookie with every request.

A new component type can be made that can be used within the Facelets view declaration language to mark another input element to remember values entered in a form.

To allow reuse it would be convenient for the developers that they can just use the functionality by adding a component to their document. JSF allows defining new component types to add functionality to the framework. This element is then an abstraction for value caching functionality.

Web storage can be used within an application written in JSF to cache data inside the browser.

By caching data in the browser, the browser presents the information directly to the user when the user performs an interaction. When the data is in the cache, there is no longer the need to send a request to the server to obtain the data, but then it can be shown directly from the cache. This will decrease the time needed to present the information, because the user doesn't have to wait for the data to arrive from the server.

A page can be rendered partially to allow caching parts of the page.

By rendering a page partially, JSF only renders a small part of the page. This allows obtaining parts of the page separately. You need this when you want to precache small parts of the page in the browser, so the script can show the information directly when the user performs an interaction.

Partial page rendering is already in the framework, but then only when the request is considered a JSF-AJAX request. JSF-AJAX requests have some side effects that are undesirable in this context. An example of such a side effect is that not all page logic is performed in such a request.

4.4.2 Websocket

A page can be made in JSF that is updated directly when information is changed by using websocket.

One of the main advantages of websocket over other techniques is to allow the client to receive updates directly, instead of polling for an update.

The websocket functionality will consist of code at both the server and the client-side. The goal of this hypothesis is to confirm that it is possible to integrate websocket into an application created with JSF.

Messages from the browser to the server via websocket can be handled through the JSF services.

Messages that arrive from the browser to the server via a websocket connection or a long polling connection don't go through the FacesServlet. Instead, they go through their own servlet. This means that none of the framework provided services are available.

When it's possible to make use of the FacesServlet, then it becomes possible to make use of the services that JSF provides. The most important service is the Facelets view declaration language, because then the complete view can be declared in a single templating language. When it isn't possible to make use of the services of JSF, then the view has to be defined with HTML in JavaScript or in another template language.

4.4.3 Canvas

The canvas-element can be used to plot the information within an application written in JSF.

To have the ability to plot the data visually will be huge asset to JSF, since JSF doesn't have a similar feature now.

An alternative to having support for plotting would be to have the possibility to render images on the server and then serve these images to the browsers. A disadvantage is that this will require more server load and that direct interaction from the user will be much harder to implement. Therefore canvas support would be a very nice to have.

The canvas-element can be used interactively in a JSF application.

Having the possibility to interact with the rendered image will increase the usability of the canvas element within JSF even further. When the scripts are able to respond to mouse events then it is possible to design rich interactions.

The mouse events that are particular interesting are the events created by moving the mouse and the events that the user creates by clicking on the canvas. Because the canvas element handles mouse move events, it becomes possible to respond when the mouse is over specific parts of the canvas. In the graph example that would be when the mouse goes over the points in the graph. With the events, it becomes possible to highlight the point the mouse is currently over.

It is possible to read data at the client-side from a DataTable component.

Because the plotting happens at the client-side, the script reads the data at the client-side. To remove the need to send the information in a way that JavaScript can understand it and a way that the user can read it, it will be more efficient when the script can also read the information directly from the table. An alternative would be to provide the data directly in a script, but then the component would send the data twice.

Currently there is no functionality like that in JSF, at least not the build-in components.

4.4.4 General

The last hypothesis is a general hypothesis that is checked for every proof of concept.

Each functionality can be offered as an extension to the framework. No changes are required in the framework.

When changes are required to the JSF framework, then it requires substantially more work to implement the feature. When it is required to change code in the framework, then the developer need to fork the code and has to maintain it separately. Therefore, it is important that the functionality can be integrated without the need to fork the original code.

5 Research

This chapter describes how the proofs of concept were developed. This chapter starts with what software we used for the proofs of concept. The implementation details for the proofs of concept follow after that. The chapter concludes with how to obtain the source code of the project.

5.1 Software used

The Java servlet container for the proofs of concept is GlassFish Open Source Edition 3.1.2. That version of GlassFish uses the reference implementation of JSF, Mojarra 2.1.7. GlassFish runs under the operating system Ubuntu 11.10. Eclipse IDE for Java EE Developers (Indigo 3.7.2) is the IDE the code is written in. PostgreSQL 9.1 was used for persistence.

JQuery 1.7.2 was used to speed up the JavaScript development. This JavaScript library aids the developer when doing DOM manipulation and performing AJAX requests.

5.2 Implementation

In this chapter will be described how the features have been designed and what trade-offs have been made.

For the proof of concept for web storage that uses local storage for caching the trade-offs are described for the choice between local and session storage, how the data is kept consistent and for the implementation of partial rendering.

For websocket is described how separation of concerns is done, how the same abstraction is made for both websocket and long polling, how the data is kept synchronized between the connected clients, how rows are identified in the table at the client and how the Facelets language is used to render the elements.

For the canvas proof of concept is described how it reads the data at the client-side, why was chosen for a component as abstraction and how other logic could be bound to click events.

5.2.1 Web storage for caching

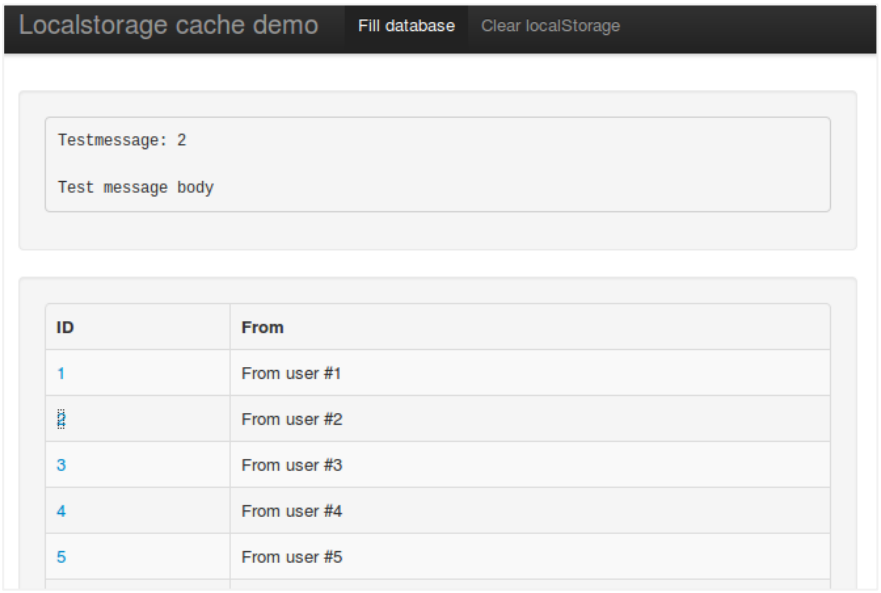

Figure 5-1: A screenshot of the proof of concept that shows how web storage is used for caching. At the bottom it shows a list of messages. When the user clicks on a message, details about the message will be shown in the top part.

Two proofs of concept demonstrate the use of web storage. The first feature is caching messages, so the user can see the message directly after an interaction. The user doesn't have to wait then for a request to the server to have finished.

For the precaching of messages, the following design has been made. A list of messages is stored in a database. To show these messages, a page shows a list with the subjects of these messages. When the user clicks on one of the message, the script prints the message body at the top. The retrieval of the message body will go through the cache.

This is a simple example that shows how web storage can be used. Techniques used for this example can also be used when you use web storage to show what products have been visited before or to add pagination to a page that instantly loads the rest of the page when the user clicks on a number.

Local storage

This proof of concept uses local storage to cache messages. This means that the messages that are cached will be still available when the user returns to the site in a new session. The alternative to local storage is session storage. The messages would then be available to the current window for the current session. When the session for the window would end, the messages are lost and have to be downloaded again. Thus local storage caches messages longer than session storage does thus it will reduce bandwidth required, reduce time needed to load the page and reduce server load.

Data consistency

A problem with caching messages longer is that the data could get out-of-sync with the server if the messages are mutable. The risk that this occurs is larger with local storage than with session storage, because local storage caches messages for a longer period than session storage. To keep the proof of concept simple this proof of concept bypasses this problem by making the data immutable. A way that this risk could be mitigated when the data would be mutable is by asking the system whether

the information is still recent. It still requires a request then, but when the information is still recent, it saves bandwidth.

Partial rendering

Because the script retrieves a single message when the message isn't in the cache, it requires partial page rendering, a way had to be found how that could be achieved. An alternative that evades this problem is to make a separate page with only the message. Requiring a separate page just for the response will make it harder to maintain, because the developer maintaining it has to keep two pages in sync.

The partial rendering implementation changes the view tree, as defined in the Facelets view declaration language, just before rendering. The component that should be rendered is put on top of the tree, together with its children. The other components from the tree are discarded. Therefore, JSF then renders only the desired element and its children.

An alternative is to replace the Writer. That replacement writer would discard the output when the components are rendered that aren't the selected component or the children of that component. This method has large overhead, because JSF renders the complete page, even though much of the output will be discarded.

Integration with JSF

The proof of concept consists of two parts. The first part is the view. The view consist of a page defined in the Facelets view declaration language and a JavaScript file. The second part is the class that provides the partial rendering functionality.

The page consists of three lines of JavaScript to initialize the script. The script calls the initialization function from the JavaScript file with as argument the identifier of the component that has to be rendered again when the user clicks on one of the links. The links call the showMessage function from the JavaScript library

The JavaScript file has 64 lines. This file contains the functions that provide the caching functionality. The script receives the messages from the server when a message isn't in the cache. The script receives the message by requesting the page with two parameters. The first parameter is the identifier of the message to be downloaded. The second parameter is the identifier of the component that should be replaced when the users clicks on one of the message links.

The class that provides partial rendering, *RenderBacking*, has 57 lines. An instance of this class adds an event listener to the event that JSF fires just before the page will be rendered. When the event is fired the instance checks whether the HTTP request contains a parameter that specifies that only a single component has to be rendered. When this is the case then the event listener discards all elements from the tree, except the single element that has to be rendered and its children.

5.2.2 Web storage for remembering values

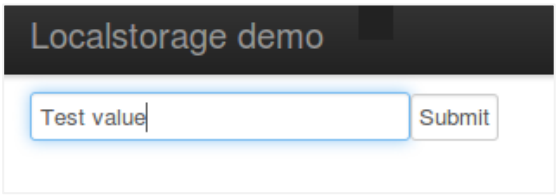

Figure 5-2: A screenshot of the proof of concept showing an input field that will remember its value when submitted

The second proof of concept shows web storage to remember the value of an input field. The values the user enters are stored in session storage. This means that the values will only be remembered for the current window. When the user opens the same page in a new window, it cannot access the information set by the already existing window. This means that it can remember separate values for each window. When the window is closed, the stored values are lost.

The code renders a piece of JavaScript to the page that does two things. First when the page is finished loading, it checks whether a value has been set for the element. When this is the case, the value of the element is set. Secondly, after checking whether a value already exists, an event listener is registered that listens for the submit event on the form the input element is in. When the user submits this form, the value is stored in session storage.

The problems encountered for this example are also encountered when you implement a web application that remembers input from the user at the client-side, for example when a web application with a shopping cart. You have to render the HTML from client-side logic and store data from the interactions in local storage.

Component

The proof of concept has a component for the Facelets view declaration language that allows marking a field as a field that should remember its value. An alternative would be to let the developer insert a small piece of JavaScript that binds the storage code to the input element.

The advantage of using a component is that the IDE can aid the developer when filling in the parameters. The tool support for writing in the view declaration language in Eclipse is better than the tool support for JavaScript in Eclipse is at the time of writing. The disadvantage is that it requires more code, because a JSF component has to be created. The code that the component will generate will call the JavaScript, thus that will have to be written anyway.

Local storage and Session storage

Since local storage and session storage have a similar interface, it would be convenient for developers to have abstractions for both of them. An attribute of the component in the Facelets view declaration language specifies the type of storage. An alternative would be to make a separate component for each type of storage. This wouldn't have many implications, because those components could then inherit from the same base class and thus would be sharing the functionality. Something that the attribute offers, but the solution with the separate component doesn't is to bind the storage type to a property in the model.

Binding

At the server-side every component has an identifier. When the component renders, it generates a HTML element that has the same identifier. This identifier is stored in the "id" attribute of the HTML element. To let JavaScript make use of the element, the script has to become aware what that identifier is. Because in HTML the identifier has to be unique, while this isn't required in the Facelets view declaration language, it is possible that the identifier of the HTML element gets a prefix. Printing out that HTML identifier is the only logic that has to be done at the server-side for web storage.

Integration with JSF

RememberInWebStorage is a component that is added to the JSF page. JSF allows adding new components by making a class in Java that extends from *javax.faces.component.UIComponent*. This class is then added to a *taglib* file. In a taglib file there is a list of all custom components that are in this project together with the attributes the components accept. In this proof of concept, there is just one component. In the top of the file is the name of the namespace. The view from the proof of concept imports that namespace at the top of the page.

The component depends on a JavaScript library, thus an annotation is added to the component indicating that a JavaScript library has to be included in the page when the page is rendered. The library is placed in the head of the page at page load. When a view has the component multiple times on the page, JSF prevents that the scripts is included multiple times into the page.

When a component renders to show the result in the browser, it outputs (X)HTML. The component has an attribute "fields" that the component uses during rendering to determine which fields need to remember its value when the form they are in submits. When the component renders, it finds the identifiers that the fields use for the "id" attribute of the rendered HTML element. The rendering starts with outputting a script element. Then script tag contains for each field that is in the "fields" attribute a JavaScript function call to register the field as a field that should remember its value.

The Java class *RememberInWebStorage* has 98 lines. The JavaScript file has 38 lines.

5.2.3 Websocket

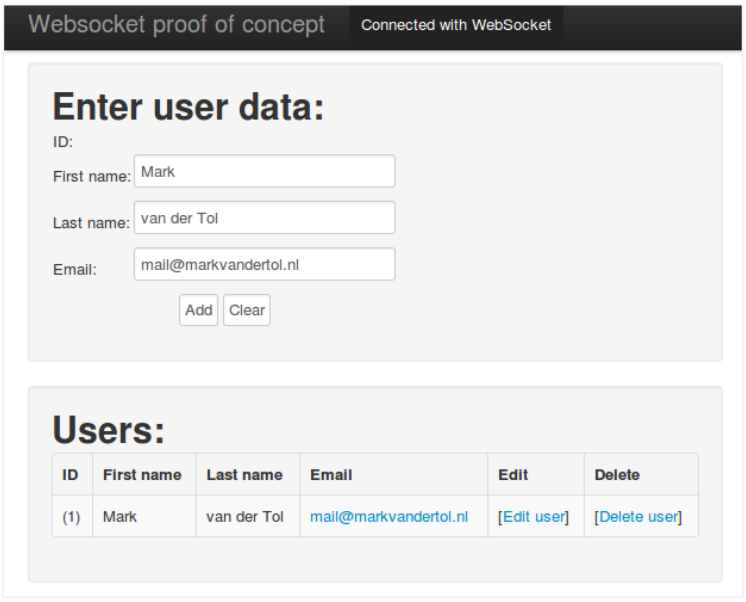

Figure 5-3: A screenshot of the proof of concept showing the user interface. The header shows how it is connected to the server. This screenshot shows that it is using websocket. At the top is the form where clients can enter new user records can or an existing user record can be edited when the client clicks on "Edit user". At the bottom is a table with the current users in the database. The table updates automatically when someone that is connected makes a change to the table.

To enable websocket functionality in a web application running in GlassFish the application has to start to listen for new websocket connections. The proof of concept uses the Comet component from the Grizzly framework to enable long polling. A newly created servlet listens for new long polling connections. To offer a consistent abstraction there has been made an abstraction on top of the websocket and long polling abstraction, so these can be accessed from a single abstraction. The communication between the server and the client uses objects that are serialized as JSON.

A page with websocket functionality loads as a normal web page in the web browser. The web page contains the table with the users currently in the database. The script in the page tries to open a connection to the server using websocket. When the connection is open, the browser can receive updates to the table from the server. When the user adds, edits or removes a user entry, also an update is sent over the socket. When the browser doesn't support websocket an attempt is made to connect to the server using long polling. When long polling is used, an AJAX request is send to the server when the page is loaded. The server will respond to that request when a record of the user table has been added or updated. When the browser has handled the response, the browser opens a new connection to wait for new updates. When the client adds, edits or removes a user entry this is sent using a separate AJAX request.

This proof of concept is an example for applications that require instant updating of values. Other examples of this could be stock tickers or collaboration applications, such as a shared virtual whiteboard. In this example, the updates of information come from other users, but the application can also be adapted to receive updates from other applications. These implementations would have to solve similar problems.

Separation of concerns

An abstraction separates the business logic from the logic that is required to setup a websocket connection and the logic that handles the changes to the table. It was done that way, because separation of concerns is considered a good engineering principle. It allows reuse of the code and makes the code easier to maintain.

The proof of concept shows a table where the client can enter user details. This logic is separated from sending over the changes. The developer has to define how the row will be rendered. Example of use in the Facelets view declaration language:

```
<m:directUpdateRepeat var="user" rowIdVar="rowId"
    value="#{index.users}" id="testTable" idField="id"
    formFields="id firstName lastName email"
    websocketUri="ws://localhost:8080/POC-Websockets/websockets"
    longpollUri="http://localhost:8080/POC-Websockets/comet"
    elementType="tbody">
    <tr data-index="#{rowId}">
        <td>(<span data-field="id">#{user.id}</span>)</td>
        <td data-field="firstName">#{user.firstName}</td>
        <td data-field="lastName">#{user.lastName}</td>
        <td>
            <a href="mailto:#{user.email}" data-field="email">#{user.email}</a>
        </td>
        […]
    </tr>
</m:directUpdateRepeat>
```

The developer has to define what collection of data he wants to render in the value attribute, what the corresponding form fields are and what the URIs are for the websocket protocol and long polling requests. The body contains the data for each row. The “data-field” attribute defines to what form field the element maps. This is needed when the edit button is clicked to populate the input elements of the form.

Same abstraction for websocket and long polling

Because not all popular browsers at the moment have support for the websocket protocol, a fallback mechanism has been made. When websocket isn’t available, a long poll connection will be setup instead.

Both the websocket code and the long poll code provide an implementation for the same abstraction, namely the possibility to send and receive messages from the browser to the server. The abstraction hides the specifics for each protocol. This is done both on the client- and the server-side.

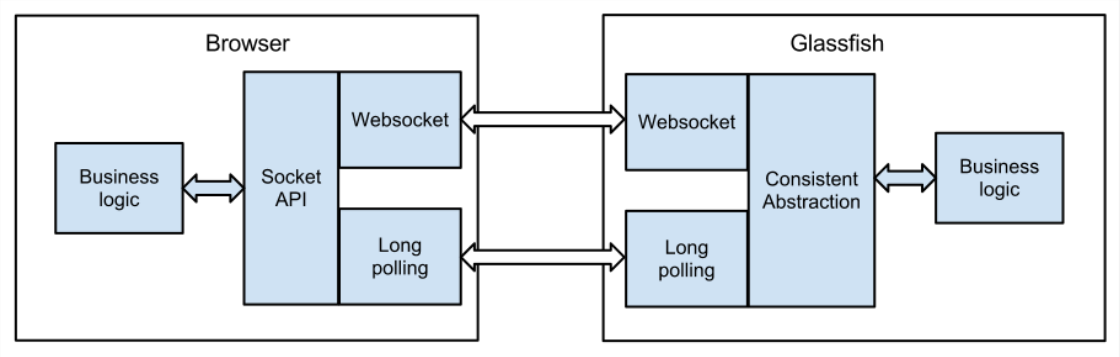

Figure 5-4: Design of the abstraction

Synchronization

When data is changed in the table by one client, this is propagated to the other clients. In "Designing Rich Internet Applications with Web Engineering Methodologies." [PRECIADO *et al.*, 2007] three ways are described to keep data synchronized. The first one that is proposed is to only send over the fields that have been changed. The second is to send over the complete row when something has changed on that row. The third alternative is to send over a part or the complete table.

Sending over the only rows that have changed fields has the advantage that is saves bandwidth when not all fields have been changed. It however requires that when a row is submitted all fields are compared to the new values. The changed values that are then sent to the connected clients.

Sending over the changed completely changed row has the advantage that the fields don't have to be compared with the previous value. When the row contains many values that haven't changed, some bandwidth is wasted.

Sending over the complete table can be simpler in some cases. It removes the need to render a single row in the Facelets view description language. The disadvantage is that a lot of bandwidth is wasted with sending parts of the table that haven't been changed. When the table grows bigger, the bandwidth that is wasted grows too.

The proof of concept sends over the changed rows. The advantage of sending individual changed fields is considered too small and sending over the complete table too inefficient.

Sending submits from browser to the server

When the user has entered the values in the form to edit or add a row, the values in the form have to be submitted to the server for persistence and to inform the listening clients about the new values. The websocket connection can also be used to send to data to the server, since a websocket connection is bidirectional. When long polling is used, this isn't a possibility, since when a request has been sent and the script waiting for a response from the server no new data can be sent to the server over that connection.

To solve the problem that no data could be send over a long polling connection that is waiting for a response from the server two alternatives were considered. The first one is using a new connection that sends the data that has to be submitted. The second alternative is to close the pending request and make a new connection. That new connection will contain the data from the form and will start waiting for updates from the server.

The advantage of the first alternative over the second is that the code for sending and receiving data can be separated in the first alternative. This isn't possible in the second alternative, since the already existing connection has to be closed and the new connection that will be made has to be aware of the data to be sent.

The disadvantage of the first alternative over the second is that the server has to handle multiple connections from the same client in the first alternative. Servers can handle multiple connections from a single host, although it could increase the load on the server. The proofs of concept uses the first alternative since it was the simplest implementation and that it scaled enough to accept the disadvantage.

Direct update table on submit or wait for event from server

When the user enters the form for a new user or to alter an existing user the data is sent to the server as described in the previous section. The table with users has to be update for all connected clients, including the table at the client who initiated the update. For this last specific case two alternatives were considered.

The first alternative is to directly update the table after the user clicks on the submit button. An advantage of this is that the table is updated instantly after the user clicks on that button. When the data sent doesn't validate at the server or when the sending the data fails, the table shows an inconsistent state.

The second alternative is to not instantly update the table after the submit button has been clicked, but wait for the server to send the event that a row has been updated. This alternative has been chosen.

Row identification

To update or remove on the page that is currently displayed in the browser the row needs to be identified. The default way in JSF to identify a row in the table is using the "clientId". The clientId of the row is the clientId of the table, appended with the index of the row starting at zero. The data model works with an identifier (primary key) that starts at one. When a row is removed from the data model the identifiers from the rows after that don't change. Therefore, the row identification of JSF isn't compatible with the identification from the data model.

Using a custom data attribute bypassed this problem. In HTML5 it is possible to add your own custom attributes to the elements.[HICKSON, 2012-1] Each row gets a custom data attribute with the identifier from the data model.

Rendering in the Facelets view declaration language

This proof of concept has the problem of rendering a single element too, just as the proof of concept for precaching web has. This proof of concept uses the same solution as that proof of concept uses.

However, to do that, another problem had to be solved first. The messages that are sent from the browser over websocket arrive on their own interface, an interface that doesn't have access to the functions of JavaServer Faces. The reason for that is that JavaServer Faces isn't initialized on that servlet.

Here is a comparison of three alternatives. The first alternative one is to setup JSF on that interface. To setup that connection an HttpRequestContext and an HttpResponseContext are needed. These

contexts are normally made by the webserver when handling the request. When a message arrives from a websocket connection, then these contexts haven't been setup. Creating such a context outside an HTTP request is hard, because such a context has many fields that need to be implemented. The second alternative is to perform a full HTTP request to make it possible to make use of the request lifecycle. The third alternative is running the JSF servlet on an embedded container. This is complicated, because this embedded container has to be setup and configured first. The second alternative is implemented in the proof of concept. An HTTP request will be send to the servlet container it is currently in.

Integration with JSF

The websocket proof of concept introduces a new component to use in the view declaration and adds an event listener for the event that triggers before the page will be rendered. The event listener is the same as in the web storage proof of concept.

In the view declaration there is a component that renders the list of users and renders the scripts to make sure that the list of users is kept in sync with the other connected clients. The component extends from *javax.faces.component.UIData*, which extends *javax.faces.component.UIComponent* as every component does. The component extends from *UIData*, because *UIData* provides services for components that render multiple rows. One of these services is that the component gets an attribute that allows binding a collection object to component. During rendering, each row from the collection will be used to render the children of the component.

The interfaces for websocket and long polling connect directly to the servlet container. Therefore, these interfaces don't integrate with JSF. The business logic makes then, after a message is sent to one of these interfaces, use of JSF by placing an HTTP request to the servlet container to render a single row. To render a single row, the same event listener as in previous proof of concept is used.

The Java files for the websocket abstraction have 711 lines in total. The JavaScript files have 268 lines in total.

5.2.4 Canvas

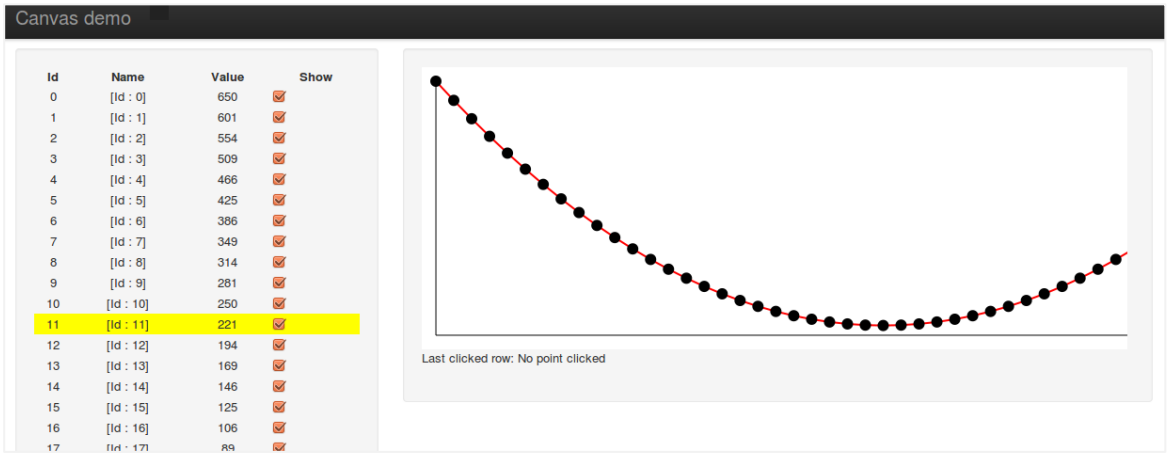

Figure 5-5: A screenshot of the canvas proof of concept. At left side is the table with the points. Points can be disabled with the checkbox in each row. The enabled points are plotted at the right side. When the user hovers over a point in the graph, the row is highlighted in the table on the left. Below the graph is information about the last clicked point. This information is retrieved with AJAX after the click.

The proof of concept shows an application that shows on the left a table with data. On the right a graph is drawn that plots that data. The user can select on the left, which rows will be visible and which rows will be hidden in the graph. The graph also responds to mouse events. When the cursor moves over a point in the table, the row that contains that point is then highlighted in yellow. When the user clicks on a point, that row will be highlighted green.

The data used in the proof of concept is a simple list of objects. Each object consists of an integer identifier and a value. When the web application starts it populates the list with objects that represent points in a formula.

The code consists of a part HTML and a part JavaScript. The server is only used to provide the data. The rendering of the canvas is separated from the business logic. The logic used for plotting the data can be reused in other applications.

This proof of concept represents applications that interactively display data from the page. Almost everything happens on the client-side, but it shows that interaction with the server-side is also possible. This proof of concept shows a graph, but alternative examples could also be interactive diagrams or a completely different way to represent data.

Reading data

In one of the hypotheses asserts that it is possible to read the data directly from the table by the script. It was shown that it was possible to read from a column in a table using the DOM API. The script identifies the right cell from the row by the HTML class attribute of the column. When the corresponding cell is found, the data from that cell is used to plot it as a point.

It would also be possible to use custom data attributes, instead of using the class attribute. This wouldn't result in any changes in the design. It would only require a small change at the point in code where the cell is selected.

Component

The code for making a graph hides behind an abstraction. For this two alternatives were considered. The first alternative that was considered was putting the required JavaScript in a file. The developer then would only have to refer to that file from page and call a function from that file. The second alternative would be to create a JSF component that hides the implementation. The component then includes the required script by itself and calls the function from that script by itself. The disadvantage would be that it requires more code to make a new component. We made a component, because a component is more convenient for a developer to use.

Binding events

The proof of concept shows that it's capable of responding to mouse events. The proof of concept shows two uses for mouse events. The first one is a way to highlight a row in the table where the points come from. The second one is to have a JSF-AJAX request after a click.

The row highlighting extends the "*ClientBehaviorHandler*" from JSF. Client behaviors generate JavaScript that respond to events in the browser. Client behaviors can be added to components in JSF. A row highlighter can be added to a canvas component by just adding a client behavior to the canvas component.

The canvas component is compatible with the build-in "*AJAXClientBehaviorHandler*". The AJAXClientBeviorHandler makes it possible to perform an action at the server when the associated event occurs at the browser and then render part of the page again. This is the already build-in way to do interactions, without having to reload the page.

After the user clicks on a point in the graph, a hidden input field is given the id of the point that is clicked as value, so it can be send to the server using a POST request when a AJAX client behavior is attached.

Integration with JSF

The canvas proof of concept integrates with JSF at two points. A component renders the canvas element and the scripts that are required for the canvas element. A client behavior is made that renders JavaScript provides the row highlighting.

The component extends from *javax.faces.component.UIInput*. The component first renders the HTML canvas element. Then the component renders then an input field. The value of the input field contains the index of the last clicked point. Because the component extends from *UIInput* and not directly from *UIComponent* the value is available at the server when the form is submitted. The component then renders the JavaScript function calls to bind the HTML elements to scripts. The class definition has 199 lines.

HighlightRow extends from *javax.faces.component.behavior.ClientBehaviorBase*. The file with the class definition has 33 lines. Client behaviors are scripts that execute at the client-side when the client makes an interaction. In this case, when the user clicks or moves the move over one of the points in the graph, the corresponding row highlights. Because the component sends the click and mouse over events to the attached clients behaviors, build-in client behaviors and client behaviors made by others work too. The proof of concept shows that the build-in AJAX client behavior handler from JSF.

The JavaScript file that is included by the component reads the points from the *DataTable* on the page, generates the graph from that data and provides the logic for the interactions with the graph. The JavaScript file has 198 lines.

5.3 Source code

The source code for the proofs of concept is publicly available through Google Code. The URL for the online project page for the code is http://code.google.com/p/html5-in-jsf-experiments/. It allows others to study in detail how the code for this project was written, to verify the code for your own and allows others to reuse the code in their own projects.

6 Analysis and conclusion

This chapter gives meaning to the results that are found during the research. Followed by that is the conclusion where an answer will be given to the main question. Finally directions for future research are given.

6.1 Analysis

During this research the goal was to give an answer to the question: “How can HTML5 technologies be used inside applications written using server-side web frameworks?” Because, as explained in the second chapter, answering the question for all server-side web frameworks is too difficult, the scope for doing research in was reduced to JavaServer Faces.

6.1.1 Hypotheses

As described earlier, the proofs of concept are for web storage, websocket and canvas functionality. This parts gives answers to the in the fourth chapter defined hypotheses.

Web storage

Two proofs of concept test the hypotheses for web storage in combination with JSF. The first proof of concept shows a form with an input field that remembers its value when the form is submitted. The first two hypotheses will be proven with this proof of concept.

The second proof of concept shows a page with a list of messages. When a message is clicked on, there will be made an attempt to read the message from local storage. If this succeeds then no request to the server is required.

Web storage can be used within an application written in JSF to remember values entered in a form.

In the proof of concept it has been shown that it is indeed possible to let input fields of a form remember values. The proof of concept does this by generating a piece of JavaScript that binds the HTML input element to the attached JavaScript library. That scripts binds to the submit event of the form the input element is in. When the page loads, the script checks whether a value already exists in the web storage. If there is a value in the web storage, the input field’s value is changed to that value. When the user submits the form, it is possible to read the value of the input field it is bound to and store the value in session storage. Therefore, this hypothesis has been proven.

A new component type can be made that can be used within the Facelets view declaration language to mark another input element to remember values entered in a user form.

It is possible to hide the script required to the interaction behind a component for the Facelets view declaration language as an abstraction. This class for this component extends from *UIComponent*. The component is registered in the taglib. Therefore, to use this functionality only the library

containing the component and the corresponding configuration files have to be added before it can be used in other projects.

Because it is possible to add the functionality as a new component to the templating language, this proves the hypothesis. This allows adding this HTML5 functionality as a reusable component. Something that is important for adoption of the functionality.

Web storage can be used within an application written in JSF to cache data inside the browser.

The proof of concept shows that it is indeed possible to cache data in the browser, so it retrieve it later instantly. The user doesn't have to wait for a request to the server, instead in can be retrieved locally. In the proof of concept the two messages before and the two messages after the currently selected message are retrieved from the server when a message is shown. Therefore, moving to the previous or the next message will be instantly.

The proof of concept uses local storage to cache the messages. When a message isn't in the storage, the script makes a request to the server for the message. The page is then rendered partially to only return that message.

The hypothesis has been proven.

A page can be rendered partially.

To make the caching functionalities more useful, it would be more efficient to have the possibility to render the page partially. With "partially" it is meant that only a specific part of the page is rendered, instead of the complete page.

This functionality was harder to implement. JSF has already build-in this functionality, but it can only be used from a JSF-AJAX request. Rendering the page from a JSF-AJAX request has some side effects. For example, the result will be decorated with XML and not all elements will be decoded. To make it possible to make use of normal AJAX requests, with normal AJAX requests is meant requests that are made without making use of the JavaScript API of JSF, a workaround was required.

The workaround is an event listener for the pre-render event. Before JSF renders the page, the component that should be rendered is put on top of tree. Components that aren't the child of that component are discarded and not rendered.

Therefore, the hypothesis is proven true, but there is a workaround required to get it working.

The functionality can be offered as an extension to the framework. No changes are required in the framework.

In the proofs of concept no code in the JSF framework had to be changed. The extensions in these proofs of concepts are formed by making components and including JavaScript files into the page. Therefore, the hypothesis is confirmed.

Websocket

The proof of concept consists of a page that shows a table of users. Above that table is a form where the user can enter the details of a new user. When the user submits a form all other users that currently have the page open will receive the update immediately.

A page can be made in JSF in combination with websocket that is updated directly when information is changed.

When the user submits the form on the page, then the server sends the new data to all connected clients. Therefore, the information updates directly when the information is changed. Thus, the proof of concept proofs the hypothesis.

The proof of concept achieves this by integrating a JavaScript file and binding the rendered HTML elements to functions in that file. The websocket component generates JavaScript that binds the rendered HTML elements to the JavaScript functions. When the page loads, JavaScript opens a websocket connection to the server. When the user submits the form, the script sends the form data through that connection. The server sends then the data to all connected clients. The script will use that data to update the table.

Messages from the browser to the server via websocket can be handled through the request processing lifecycle.

To make it possible to define the layout of the elements that need to be rendered again after an update of the data in the Facelets view declaration language, it is required that the request processing lifecycle is run. The problem is that messages from the browser sent via a websocket connection don't go through the lifecycle, but arrive at a different interface. Because JSF isn't set up on that interface, it cannot be run directly from there.

A workaround makes it possible to make use of the request processing lifecycle from that separate interface anyway. This is done by doing an HTTP request to the servlet container. The servlet container sends the HTTP request then trough the FacesServlet.

Therefore, it is possible to make use of the request processing lifecycle, although a workaround is required. This proves the hypothesis.

The functionality can be offered as an extension to the framework. No changes are required in the framework.

Also in this proof of concept, no code in the JSF framework had to be changed. The code for in the view could be provided by creating a new component. Therefore, all the code written is an extension to the framework. Therefore, the hypothesis is true.

Canvas

For the canvas feature a page is created that shows on the left a list of points and on the right a graph. In the proof of concept the data from the list is plotted in a graph. The two axes are drawn, the points are drawn and lines between the points are drawn.

The canvas-element can be used to plot the information within an application written in JSF.

A new component renders the HTML and the scripts required to make a graph with the canvas element. The script that reads the points draws those into the canvas element. JavaScript is generated to bind the functions in the JavaScript file to the HTML elements.

Because it shows that information can be plotted, the hypothesis has been proven.

The canvas-element can be used interactively in a JSF application.

In the proof of concept it is shown that the JSF application can respond to mouse events. This is one way to make it interactive. When the user puts his cursor over one of the points, the row for the point is highlighted in the table.

This is done by making the canvas component compatible with client behaviors. These behaviors perform when the events that they are attached to happens at the client-side. The row highlighting is implemented as client behavior.

Another way that shows that the component can be used interactively is that points can be disabled in the table. The graph directly updates when a point is removed or added, because the script listens events on these checkbox elements.

It is possible to read data at the client-side from a DataTable component.

To remove the need to send over the data twice (once for in the table and once for the graph) the script should be able to read the data directly from the table. The proof of concept shows this.

The scripts reads the points from the table when the page loads. The class attribute of the cells in the table tell the script what the value is of this column.

The proof of concept shows that it is possible to this, therefore the hypothesis has been proven.

The functionality can be offered as an extension to the framework. No changes are required in the framework.

Also in the proof of concept for the canvas functionality no code in the JSF framework had to be changed. The canvas functionality could be offered by making a new component. Therefore, all the functionalities that were added were added as extensions to the framework. Therefore, the hypothesis has been proven.

6.1.2 Design

In this section the most important design characteristics are given. These are the things that have to be kept in mind when similar features are implemented in the future.

Common design issues

For some features the most important part consisted of adding snippets of generated JavaScript to the page. Those snippets of JavaScript bind scripts with the functionality to elements on the page. For all the considered features logic at the server-side was required.

For all the features, it was also possible to hide the implementation behind an abstraction, so that the functionality can be reused. JSF offered components to make the abstractions available in the view declaration language. Client-side interactions were hidden behind client behaviors.

Web storage

The most important characteristics of the web storage implementation are that logic is required at the server and the client-side, and that at the server the possibility had to be added to render the page partially. For the server-side it was needed to generate the script that calls the library, the component identifier has to be generated.

To allow precaching small parts of the page, it required to have the possibility to receive only a certain part of the page. This was implemented by changing the component tree just before rendering. This can in certain case have some side-effects. A cleaner approach would be to have this functionality in the framework directly.

Websocket

The websocket communication logic can be separated from the business logic. This will improve the maintainability and the reusability. The reason that logic was required at the server-side is that a new connection handler had to be added to the server. Also extra business logic is needed at the server-side. At the client-side JavaScript was needed to connect to that connection handler and logic to send and receive data over that connection.

Data has to be kept synchronized between the server and clients. A tactic has to be picked on how the data is synchronized. The proof of concept keeps the data synchronized by sending over the changed rows to all clients. Sometimes it is more convenient to send over the complete data or sometimes it is more efficient to only send over the changed fields of the row.

Having the possibility to use a single templating language to define the view, even when the parts of the view will be requested later by JavaScript, is very convenient for the developer. In JSF that template language is the Facelets view declaration language. There was a workaround needed to make use of the request processing lifecycle, making it possible to render a part of the page. An HTTP request had to be used to have access to the request processing lifecycle.

Canvas

Most functionality is at the client-side. Only generating the scripts to bind the HTML elements to the scripts is done at the server-side.

Reading data from the page can be done without any problems when JavaScript is used. Thus sending over data in HTML form and in script form isn't required.

JSF offers functionality to add JavaScript to client-side events. Therefore adding these is straightforward.

6.1.3 Validation

Two developers who have developed with JSF for many years validated the code. They have looked into the code and validated that JSF was used correctly. They also helped by discussing the design for the functionalities.

Care has been taken to prevent extrapolating too far with the results that were obtained during this project. There has been considered what the differences are between the researched environment and the whole environment the question describes.

The line counts show that the functionalities we build are on top of JSF and that the code of JSF isn't forked.

6.2 Conclusion and discussion

The main question was: How can HTML5 technologies be used inside applications written using server-side web frameworks? Because every client-side HTML5 technology that was researched

could be integrated into a web application based on JSF, it can be said that JSF can be used for these client-side HTML5 technologies. The functionalities that are similar to canvas, websocket and web storage can be used within a JSF application.

This means an organization that is using JSF to build their web applications can continue to use JSF and while still being able to access the new functionality HTML5 offers. This was an important use case to start this research for. Organizations can adopt existing pages or add new pages that make of HTML5 technologies, while other pages don't have to be touched. Starting a new project using JSF and HTML5 was also an important use case. The proofs of concept prove that this is possible.

When we extrapolated the results to other server-side web frameworks, the differences between those server-side web frameworks were taken into account. There are many different server-side frameworks, with different design philosophies. Therefore, functionalities that only require adding scripts or simple elements will probably work fine in other server-side frameworks, but other functionalities could be harder, if possible at all, to implement.

6.2.1 Advantage of using server-side frameworks

One of the reasons to integrate HTML5 technologies into JSF was to make it possible to keep using the same way of developing of web applications. That JSF can be successfully used is clear when you look at the number of web applications made with JSF. Companies that are already using JSF, or another servers-side framework, can continue to use these frameworks. It is possible to introduce new HTML5 features on one page, while not changing the other pages. Using a server-side framework has other advantages too. Because the focus is on the server-side the developer has to deal with less client-side issues.

The components created during this research integrate the client- and the server-side development. Developers can add functionality in the form of abstractions. These abstractions deal with the client-side logic and the communication between the server- and the client-side.

By hiding the functionality behind components, the developer doesn't have to deal with client-side development. The developer doesn't have to work in another programming language, such as JavaScript, but can make the complete application with Java.

6.2.2 How to integrate HTML5 into JSF

The message caching proof of concept shows that it is possible to make use of JSF to introduce HTML5 functionality without creating a new component. The trade-off is that it is less work calling the JavaScript library directly compared to creating a component, but less implementation details are behind an abstraction. The developer has to include the depended JavaScript library himself then, while a component can do this for the developer.

The other proof of concepts hide the functionality behind a newly defined component. This research shows that it's possible to make new components, which hide the HTML5 functionalities behind an abstraction. These components can then be used in the view declaration. Developers using these components don't have to deal with the client-side interactions anymore. They can quickly bind these components to server-side logic. The component is responsible making these interactions work.

JSF provides various abstract classes that form the basis for new components. For a simple component that doesn't render rows and doesn't accept input from the user, *UIComponent* can be

used. For components that render a collection of rows, *UIData* is more suitable. *UIInput* is more suitable for components that render a form field, because they can handle input from the client.

JSF adds the JavaScript libraries where the components depend on to the head of the page when JSF renders the view. The component themselves can also output pieces of JavaScript. The components in the proofs of concept output JavaScript to bind generated HTML elements to the scripts. To generate scripts that respond to events at the client site Client Behaviors are the way to go in JSF. Developers using these components don't have to write the JavaScript themselves. The components generate it for them.

Partial rendering is a problem that occurs at two of the proofs of concept. Partial Rendering is rendering only a small part of the page. Web applications that require this are web applications that use single page interfaces. That is a page where a part of the page has to be updated, but navigating to a new page isn't desirable. Partial rendering is included in JSF for use with the build-in AJAX Client Behavior handler. It isn't possible to make use of the build-in partial rendering of JSF without performing an AJAX postback. One workaround, which is used in the proofs of concept, is to change the component tree just before rendering so that only the selected component with its children will render. When a single page interface is made and partial rendering isn't available in the used framework, a new server-interface has to be defined to communicate with the browser and the server. This interface has to send the data and the browser has to interpret that data and update the page accordingly.

The websocket proof of concept shows that it is possible to make use of the services JSF provides from requests that don't go through the FacesServlet. The proof of concept needs it because the websocket connections go through a different interface. The websocket handler code sends an HTTP request to the servlet container it is in. This adds overhead. It is more efficient when the services of JSF are initialized in the websocket servlet, but this is more complex. When a framework is used to cannot offer its functionalities to other interfaces, it is hard to make an application that makes use of multiple protocols for communication. Communication between the interfaces for the protocols is then needed.

6.3 Contribution

This research shows that it is possible to develop web applications that make use of HTML5 with a server-side web frameworks and this research shows how it can be done. This is useful for the following cases:

- To determine how to implement HTML5 technologies into a server-side framework.
- When selecting a framework to develop a web application that will make use of HTML5 technologies in, the findings can be used to determine whether a server-side framework is suitable.

Because the focus in this project is on JSF, this thesis gives detailed information on how HTML5 functionalities could be integrated in that server-side framework. When integrating HTML5 functionalities into another server-side framework, then there has to be determined whether the concepts applied in these proofs of concept can also be applied to that framework.

6.4 Future research

The answer to the main question raises other new questions. The answer to the main question doesn't give a clear answer what this means for other web frameworks. Another point of interest would be how all logic could be moved to the client, thus making a server-side application a client-side application. Finally, research on how providing rich interactions improve the usability would be an interesting point for further research.

6.4.1 How to standardize

To make the use of HTML5 functionality more convenient for the developers good abstractions are required. Before good abstractions can be made it has to become clear what common use cases are for these functionalities. Then has to be identified what these use cases have in common and what parts of the code could be reused if they were abstracted.

After abstractions have been found, developers can start using these abstractions. After some time there can be confirmed whether these abstractions are good abstractions. These good abstraction could then be standardized. Research would have to start with identifying what the common use cases are that could benefit from an abstraction.

6.4.2 Other server-side frameworks

During this project only JSF was considered. As described in previous paragraphs it is hard to extrapolate the results to other web frameworks. Therefore, it would be interesting to see whether the results obtained in during this project can be replicated in other server-side web frameworks. It would give an answer whether the choice of framework matters when you want to make use of HTML5 functionality.

6.4.3 Moving all logic to the client

Another subject for further research would be how to migrate an existing web application that is made with a server-side web framework to a web application that performs almost all logic in the client, making it almost a client-side application. The server would then only be needed for the retrieval of data and the persistence of data. During this project is researched how HTML5 functionalities could be added to existing web applications, because companies have invested heavily in these kinds of applications. When the migration of moving all the logic client-side can be made straight forward, then this would be a good alternative.

6.4.4 Usability

During this project no extensive research was done in usability. Do the pages really improve when the new HTML5 functionalities are used? Developing web applications that make use of rich interactions take more time to develop than applications that don't make use of these kinds of interactions. In addition, the load time of pages with a lot of client-side logic can be much higher than pages that do that same logic on the server. Having an answer to the question whether it pays off to make use of rich interactions can aid organizations that are thinking about upgrading their site or when they are at the start of a new project.

An interesting case where it is shown that client-side business logic isn't always more efficient than server-side business logic is the Twitter update in May 2012. All the rendering was done before that update in the browser. By moving the rendering logic to the server, the pages load 80% quicker.[TWITTER, 2012]

Bibliography

ANKOLEKAR, A., KRÖTZSCH, M., TRAN, T. AND VRANDECIC, D. 2007. The two cultures: mashing up web 2.0 and the semantic web. In *Proceedings of the 16th international conference on World Wide Web*, 825-834.

APACHE MYFACES. 2012. Why should I use MyFaces Core? *Apache MyFaces FAQ*. https://cwiki.apache.org/confluence/display/MYFACES/FAQ#FAQ-WhyshouldIuseMyFacesCore%3F - Retrieved June 12 2012.

BERNERS-LEE, T. 1996. WWW: past, present, and future. *Computer* 29, 69-77.

BRUTLAG, J. 2009. Speed Matters for Google Web Search.

BURNS, E. AND SCHALK, C. 2010. JavaServer Faces 2.0 The complete Reference. Mc Graw Hill, ISBN: 0071625097.

DANIEL, F. 2007. Understanding UI Integration: A Survey of Problems, Technologies, and Opportunities. *IEEE Internet Computing* 11, 59-66.

ECMA. 2011. ECMAScript Language Specification Version 5.1. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

FARRELL, J. AND NEZLEK, G.S. 2007. Rich Internet Applications The Next Stage of Application Development. In *ITI '07 Proceedings of the 29th International Conference on Information Technology Interfaces*, 413-418.

FIELDING, R.T. AND TAYLOR, R.N. 2002. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology* 2, 115-150.

HICKSON, I. 2011. Web Storage. http://www.w3.org/TR/webstorage/ - Retrieved June 12 2012.

HICKSON, I. 2012-1. 3.2.3 Global attributes – A vocabulary and associated APIs for HTML and XHTML. *W3C*. http://www.w3.org/TR/html5/global-attributes.html - Retrieved June 14 2012.

HICKSON, I. 2012-2. HTML Canvas 2D Context. *W3C*. http://www.w3.org/TR/2dcontext/ - Retrieved Jun 12 2012.

HICKSON, I. 2012-3. The canvas element — HTML5. *W3C*. http://www.w3.org/TR/html5/the-canvas-element.html - Retrieved May 31 2012.

HICKSON, I. 2012-4. The WebSocket API. *W3C*. http://www.w3.org/TR/websockets/ - Retrieved June 14 2012.

JAVA COMMUNITY PROCESS. 2004. JSR 127: JavaServer Faces. http://www.jcp.org/en/jsr/detail?id=127 - Retrieved June 12 2012.

JAVA COMMUNITY PROCESS. 2008. JSR 252: JavaServer Faces 1.2. http://www.jcp.org/en/jsr/detail?id=252 - Retrieved June 12 2012.

JAVA COMMUNITY PROCESS. 2010. JSR 314: JavaServer Faces 2.0. http://www.jcp.org/en/jsr/detail?id=314 - Retrieved June 12 2012.

JAZAYERI, M. 2007. Some Trends in Web Application Development. In *FOSE '07 Future of Software Engineering,* 199-213.

JOBS, S. 2010. Thoughts on Flash. *Apple*. http://www.apple.com/hotnews/thoughts-on-flash/ - Retrieved February 3 2012.

MESBAH, A. AND VAN DEURSEN, A. 2007. An Architectural Style for AJAX. In *WICSA '07 Proceedings of The Working IEEE/IFIP Conference on Software Architecture,* 44-53.

MIKKONEN, T. AND TAIVALSAARI, A. 2008. Web Applications - Spaghetti Code for the 21st Century. In *SERA '08 Proceedings of the 6th International Conference on Software Engineering Research, Management and Applications,* IEEE Computer Society, Washington, DC, USA, 319-328.

NAH, F.F. 2004. A study on tolerable waiting time: how long are Web users willing to wait? *Behaviour & Information Technology* 23, 153-163.

NIELSEN, J. 1993. Response Times: The 3 Important Limits. http://www.useit.com/papers/responsetime.html - Retrieved June 12 2012.

PAUL, R. 2007. The security risks of AJAX/web 2.0 applications. *Network Security* 2007, 4-8.

PRECIADO, J.C., LINAJE, M., COMAI, S. AND SANCHEZ-FIGUEROA, F. 2007. Designing Rich Internet Applications with Web Engineering Methodologies. In *Web Site Evolution, 2007. WSE 2007. 9th IEEE International Workshop on,* 23-30.

TAIVALSAARI, A., MIKKONEN, T., INGALLS, D. AND PALACZ, K. 2008. Web Browser as an Application Platform. In *SEAA '08 Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications,* 293-302.

TWITTER. 2012. Improving performance on twitter.com. *Twitter Engineering Blog*. http://engineering.twitter.com/2012/05/improving-performance-on-twittercom.html – Retrieved June 20 2012.

VAN KESTEREN, A. 2012. XMLHttpRequest Level 2. *W3C*. http://www.w3.org/TR/XMLHttpRequest/ - Retrieved June 5 2012.

VAUGHAN-NICHOLS, S.J. 2010. Will HTML 5 Restandardize the Web? *Computer* 43, 13-15.

WHATWG. 2012. HTML Standard - Session history and navigation. http://www.whatwg.org/specs/web-apps/current-work/multipage/history.html - Retrieved March 8 2012.

WHEN CAN I USE. 2012. When can I use... Web Sockets. http://caniuse.com/#search=websocket - Retrieved June 12 2012.

Appendix A: Implementation

This appendix gives the detailed design and implementation details for the proofs of concept.

Web storage for caching

The web storage code consists of the following packages with the following classes:

- Backing: Consists of objects that provide business logic that are bound to the view.
  - MessageBacking: Provides the model for messages.xhtml. Consists of methods to receive messages from the data access object.
  - RenderBacking: Is loaded when the web application starts. It adds an event listener for the pre-render event. During the pre-render event it will determine whether a partial rendering is desired. This is done by looking at the parameters send with the request. When a partial rendering is desired, then the tree that will be rendered is adapted to make the desired component root of the tree.
- Dao: Consists of data access objects, classes that allow connecting to a database.
  - MessageDAO: Provides methods to add and read message objects from the database.
- Entities: Objects that represent rows in a database table and objects related to these objects.
  - Message: Represents a message entity in the database.
  - MessageConvertor: Converts the message object into an identifier that can be used as a parameter for a URL.

The view for the web storage proof of concept is defined in messages.xhtml. The most important code from that file is:

```
[…]
<!--Use message identifier in the request parameters to identify the current
    request -->
<f:metadata>
    <f:viewParam name="message" value="#{messageBacking.currentMessage}"
        converter="#{messageConvertor}" />
</f:metadata>

[…]

<!-- Render a single message: -->
<h:panelGroup layout="block" class="well" id="messageTarget"
    binding="#{components['message']}">
    <c:if test="#{empty messageBacking.currentMessage}">
        Click on a message below to start
    </c:if>
    <c:if test="#{!empty messageBacking.currentMessage}">
        <pre>#{messageBacking.currentMessage.message}</pre>
    </c:if>
</h:panelGroup>
```

```
<!-- Bind the PanelGroup as target to render the messages -->
<script>
    $(function () {
        message.setTargetDiv("[id='#{components['message'].clientId}']");
    });
</script>

<!-- Table with all the messages -->
<h:dataTable var="message" value="#{messageBacking.messages}"
    styleClass="table table-striped table-bordered">
    <h:column>
        <f:facet name="header">ID</f:facet>
        <a href="JavaScript:message.showMessage(#{message.id});">
            #{message.id}</a>
    </h:column>
    <h:column>
        <f:facet name="header">From</f:facet>
            #{message.fromName}
    </h:column>
</h:dataTable>

[…]
```

Web storage for remembering values

- Component: Consists of reusable components that can be used inside the Facelets view declaration language.
    - RememberInWebStorage: A Faces component that renders the required script during the rendering phase.

The view for the web storage proof of concept is defined in index.xhtml. The most important code from that file is:

```
[…]

<h:form>
    <m:rememberInWebStorage fields="field1" persistence="session" />

    <h:inputText id="field1" />
    <h:commandButton action="index" value="Submit" />
</h:form>

[…]
```

The rememberInWebStorage element will become rendered as a script. That script will add an event listener to the form it is placed in. On submit it will store the value of the fields that are in the fields attribute.

Websocket

The websocket code consists of the following packages with the following classes:

- Backing: Consists of objects that provide business logic that are bound to the view.

    - Index: Provides the model for index.xhtml. Allows index.xhtml to read a list of users.
    - RenderBacking: Is loaded when the web application starts. It adds an event listener for the pre-render event. During the pre-render event it will determine whether a partial rendering is desired. This is done by looking at the parameters send with the request. When a partial rendering is desired, then the tree that will be rendered is adapted to make the desired component root of the tree.
    - WebSocket: Provides the business logic for the websocket connection. Receives the messages from the clients and sends the updated rows back to the clients.
- Dao: Consists of data access objects, classes which allow connecting to a database.
    - UserDAO: Provides access to the Users in the database.
- DirectUpdate: The infrastructure required to provide websocket and long polling connections. This doesn't contain the business logic.
    - DirectUpdate: The façade for DirectUpdate. Provides methods to setup a connection, allows adding listeners for incoming messages and sending messages to all connected clients.
    - DirectUpdateAdapter: An interface that defines an abstraction for the connection to the clients. Used to provide a consistent abstraction for both websocket and long polling connections.
    - DirectUpdateCometServlet: Used to provide a connection interface for long polling connections. Implements DirectUpdateAdapter.
    - DirectUpdateMessageListener: Provides an interface that allows listening for incoming messages. These messages come from the client. This interface is to be implemented by the business logic that wants to deal with these messages.
    - DirectUpdateRepeatComponent: Provides a Faces component that can be used in the view. This component takes care of rendering all the rows of the connected data set and outputs the JavaScript that calls.
    - DirectUpdateTextMessageListener: Listens for text messages from the clients. The messages are sent over websocket as text. These messages have to be converted to an instance of IncomingMessage.
    - DirectUpdateWebSocketApplication: Used to provide a connection interface for websocket connections. Implements DirectUpdateAdapter.
    - HttpRequestHelper: A convenience class to help with making HTTP requests. This is used to make it possible to make requests to the request lifecycle.
    - IncomingMessage: Entity class in which messages arrive from the clients.
    - OutgoingMessage: Entity class in which messages are sent to the clients.
- Entities: Objects that represent rows in a database table and objects related to these objects.
    - User: Represent a row in table User.
    - UserConvertor: Converts a user object into an identifier that can be used as request parameter and back.
- Util: Convenience classes that don't fit anywhere else.
    - JavaScriptHelper: Provides static methods that help with generating JavaScript functions.
    - StringUtil: Some generic string functions for convenience.

The static files are index.xhtml, websockets.js and directUpdate.js. Index.xhtml contains the view for the proof of concept. Websockets.js provides an abstraction over the websocket connection. This

abstraction includes a fallback to long polling when websocket isn't available. DirectUpdate.js uses that abstraction to create an automatically updating table.

The most import content of index.xhtml is shown below.

```
<m:directUpdateRepeat var="user" rowIdVar="rowId" value="#{index.users}"
    id="testTable" idField="id" formFields="id firstName lastName email"
    websocketUri="ws://localhost:8080/POC-Websockets/websockets"
    longpollUri="http://localhost:8080/POC-Websockets/comet"
    elementType="tbody">
       <tr data-index="#{rowId}">
          <td>(<span data-field="id">#{user.id}</span>)</td>
          <td data-field="firstName">#{user.firstName}</td>
          <td data-field="lastName">#{user.lastName}</td>
          <td>
             <a href="mailto:#{user.email}">
                <span data-field="email">#{user.email}</span>
             </a>
          </td>
          <td>
             [<a href="JavaScript:directUpdate.editRow(#{user.id})">
                Edit user
             </a>]
          </td>
          <td>
             [<a href="JavaScript:directUpdate.deleteRow(#{user.id})">
                Delete user</a>]
          </td>
    </tr>
</m:directUpdateRepeat>
```

Canvas

The websocket code consists of the following packages with the following classes:

- Backing: Consists of objects that provide business logic that are bound to the view.
  - DataInitialize: When the application loads, it fills the database with points, so that the application directly has to data to plot.
  - IndexBacking: Provides the model for index.xhtml. Allows index.xhtml to read a list of data points from the database.
- Component: Consists of reusable components that can be used inside the Facelets view declaration language.
  - CanvasGraph: A component that renders the canvas element on the page and outputs a piece of JavaScript that binds the components to the scripts.
  - HighlightRow: A behavior handler that generates JavaScript to deal with user interactions at the client-side. When the user mouse over or clicks on a point a row is highlighted in the configured color.
- Dao: Consists of data access objects, classes which allow connecting to a database.
  - DataPointDAO: Provides access to the DataPoints in the database.
- Entities: Objects that represent rows in a database table and objects related to these objects.
  - DataPoint: A class describing a point in a graph.
- Util: Convenience classes that don't fit anywhere else.
  - StringUtil: Some string manipulation functions.

The other files that are included into this proof of concept are index.xhtml and canvas.js.

The most important part of index.xhtml is:

```
<h:form>
    <m:canvasgraph id="graph" table="dataPointTable" idColumn="id"
    valueColumn="value" visibleCheckbox="show" width="1000" height="300"
    lineColor="red" value="#{indexBacking.value}">
        <f:ajax event="click" render="testoutput" />
        <m:highlightRow event="mousemove" cssClass="highlightrow_mouseover" />
        <m:highlightRow event="click" cssClass="highlightrow_click" />
        Canvas is not supported in your browser.
    </m:canvasgraph><br />

    Last clicked row:
    <h:outputText id="testoutput" value="#{indexBacking.value}" />
</h:form>
```

The CanvasGraph component renders the canvas element and outputs the scripts that read from the table and plot the data from the table.

The AJAX element is the already build in AJAX client behavior handler. When the user clicks on the canvas, the script looks whether a point has been clicked. When the user clicks on a point the value of a hidden input field is changed to correspond to the identifier of the clicked point. Then the form submits and the element in the “render” attribute renders again.

The HighlightRow elements change the CSS class of the rows that are clicked or where the mouse goes over.

Canvas.js contains the logic for reading the data directly from the HTML table, plotting the graph and highlighting the row in the table when the mouse goes over or when the user clicks on a point in the graph.