FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

RESTful web applications with reactive, partial server-side processing in Java EE

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Jakob Korherr
Matrikelnummer 0925036

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Mag. Dr. Schahram Dustdar
Mitwirkung: Univ.Ass. Dipl.-Ing. Michael Vögler, BSc

Wien, 23.11.2015

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

RESTful web applications with reactive, partial server-side processing in Java EE

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Jakob Korherr
Registration Number 0925036

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Mag. Dr. Schahram Dustdar
Assistance: Univ.Ass. Dipl.-Ing. Michael Vögler, BSc

Vienna, 23.11.2015

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Jakob Korherr
Rötelsteig 10, 8037 Zürich, Schweiz

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Acknowledgements

I want to thank everyone who helped to complete this thesis and the associated project.

Many thanks to my advisors, especially to Michael Vögler, who greatly supported me through the course of writing this thesis, and who provided lots of valuable feedback, even on short notice and during weekends. Michael also helped in structuring and proof-reading the thesis, and he guided me through the tedious processes involved with writing a Master's thesis. Furthermore, he was very flexible, never pressured me, and assisted me in completing the thesis remotely from Zurich. I very much appreciate his understanding and easy-going nature.

My deepest gratitude goes out to my girlfriend Stefanie, who very much supported me in completing this thesis, and who always encouraged me to carry on the work in times of procrastination and lack of motivation.

Special thanks to my former colleagues at IRIAN Solutions in Vienna, who helped me in pursuing my studies while also working part-time.

Many thanks also to David Bittermann, who went on errands for me in Vienna, because I was in Zurich and unable to do it myself.

Finally, I want to thank my family and friends. Thanks for always being there for me!

Abstract

Classic web applications are sending too much data from the web server to the client browser, because they are using full page requests, although only parts of the UI are actually changing. This implies that resources on the web server and network traffic are wasted, and furthermore the user experience suffers, because of longer response times. In fact, it is sufficient to only re-render those parts of the current web page, which are actually changing, whereas the other parts of the page can remain unchanged. This approach, called single-page web applications, can be implemented using AJAX, unfortunately, introducing problems like decreased bookmarkability and reduced back-button support. However, by introducing the HTML5 history API, these problems can be circumvented.

This thesis aims to add declarative support (i.e. using special syntax rather then having to call the API manually) for using the HTML5 history API with Java EE by developing a novel web application framework, built on existing Java EE standards, for creating single-page web applications, called *Mascherl*. This framework employs partial-page rendering using AJAX in combination with the history API, in order to make the resulting single-page web application bookmarkable, and to allow the browser's back- and forward-buttons to work as expected. The framework uses server-side rendering with partial model evaluation for the partial-page updates, as opposed to client-side rendering used in popular JavaScript single-page web application frameworks like AngularJS, EmberJS, or Google Web Toolkit (GWT).

Moreover, this thesis also addresses the scalability problem of prevalent web frameworks, which mainly exists because of the usage of stateful servers and synchronous, blocking request processing. Using the resulting framework, it is possible to create RESTful single-page web applications, which run on stateless Java EE servlet containers, and can therefore be horizontally scaled without further considerations. In addition, the framework has built-in support for asynchronous, non-blocking request handling, which allows the web application server to serve a great number of concurrent requests without blocking.

On top, this thesis presents an example application, which is built using the developed framework. This example application is then used for evaluating the framework's performance.

Kurzfassung

Klassische Webapplikationen senden zu viele Daten vom Webserver zum Browser des Benutzers, weil für jede Server-Anfrage eine vollständige Seite generiert und übertragen wird, obwohl sich nur Teile dieser Seite wirklich ändern. Diese Verschwendung von Server- und Netzwerk-Ressourcen führt zu einem verminderten Nutzererlebnis, aufgrund der längeren Anfrage- und Übertragungszeiten. In Wirklichkeit reicht es vollkommen aus nur die Teile der aktuellen Seite neu zu berechnen und zu übertragen, welche sich auch tatsächlich ändern, wohingegen alle anderen Teile der Seite unverändert bleiben können. Dieser Ansatz, welcher unter dem Begriff "single-page Webapplikationen" bekannt ist, kann mit Hilfe von AJAX implementiert werden, wobei das jedoch zu neuen Problemen führt, wie z.B. schlechterer Unterstützung für direkte Links zu jeder beliebigen Seite der Webapplikation oder verminderter Funktionalität des Zurück-Buttons des Browsers auf der Webseite. Durch die Verwendung der HTML5 History API können diese Probleme jedoch beseitigt werden.

Das Ziel dieser Arbeit ist es die HTML5 History API auf deklarative Weise (d.h. mittels spezieller Syntax anstatt expliziter Aufrufe der API) in Java EE einzubinden. Realisiert wird dieses Ziel durch die Entwicklung eines neuen Webframeworks für die Erstellung von single-page Webapplikationen, basierend auf etablierten Java EE Standards, genannt *Mascherl*. Dieses Framework nutzt partielles Rendering zusammen mit AJAX und der HTML5 History API, damit die resultierende Webapplikation bookmarkfähige Internetadressen verwendet sowie die Zurück- und Vorwärts-Buttons des Browsers wie erwartet unterstützt, obwohl rein technisch der Webbrowser immer auf derselben Webseite bleibt. Das Berechnen und Rendern der partiellen Aktualisierungen der Webseite geschieht dabei vollständig am Server, im Unterschied zu bekannten Vertretern von "single-page Webapplikationen", die auf JavaScript basieren, wie z.B. AngularJS, EmberJS, oder dem Google Web Toolkit (GWT).

Außerdem beschäftigt sich diese Arbeit mit dem weit verbreiteten Skalierbarkeitsproblem von aktuellen Webapplikationen, welches hauptsächlich durch die Verwendung von zustandsorientierten Webservern sowie synchroner, blockierender Anfragebearbeitung entsteht. Mit dem neu entwickelten Webframework ist es möglich Webapplikationen zu erstellen, welche dem REST-Paradigma genügen und die Installation auf einem zustandslosen Java EE Server erlauben. Eine horizontale Skalierung dieser Applikationen ist somit ohne Weiteres möglich. Zusätzlich können Benutzer-Anfragen auch asynchron und ohne Blockieren eines Server-Threads abgearbeitet werden, wodurch eine große Anzahl von Anfragen gleichzeitig bedient werden kann.

Darüber hinaus wird die Zweckmäßigkeit das Webframeworks anhand einer Beispielapplikation präsentiert. Diese Applikation wird weiters für die Evaluierung der Performanz des Webframeworks herangezogen.

Contents

CHAPTER 1

Introduction

1.1 Motivation

In the classic web application model, full page requests are used in order to display changes on the user interface (UI). This means that for every change in the UI, a new HTTP request needs to be sent to the server, where the full page containing the UI change is created, and then sent back to the client for display [44]. Many times the server sends back a completely different page from the one that was shown before, however, most of the time the new page differs from the previous one only marginally [140].

The HTML responses of subsequent HTTP requests of the same web application oftentimes share the same resources (CSS and JavaScript files) and the same metadata in the head section of the HTML page (meta tags, title, etc), and furthermore many parts in the body of the HTML page are equal [102]. This can be explained by the usage of page templates on the server, which comprise the common parts of the HTML pages of the web application [86].

Aside from that, web applications often only need to change one part of the UI, whereas the rest of the page can stay the same [140]. Figure 1.1 shows an example from GitHub [55]. The web page contains general data about a user, and a tab pane to display different kinds of detailed information about the user. If a user clicks the link to switch a tab, the only thing that changes on the web page is the content of the tab pane, however, due to using full page requests, this small change in the UI forces the whole web page to be re-rendered, with only minor differences in the resulting HTML.

In order to avoid using full page requests every time new data has to be fetched from the server, the AJAX web application model was introduced [44]. The model allows loading data asynchronously from the server, and calling a JavaScript callback function with the received data afterwards. This callback function can then process the new data, and alter the browser DOM to enforce UI changes [105]. Experiments [1, 23, 120] show that using AJAX for partial page updates can result in 56% - 73% smaller response sizes, and thus a much better responsiveness of the web application.

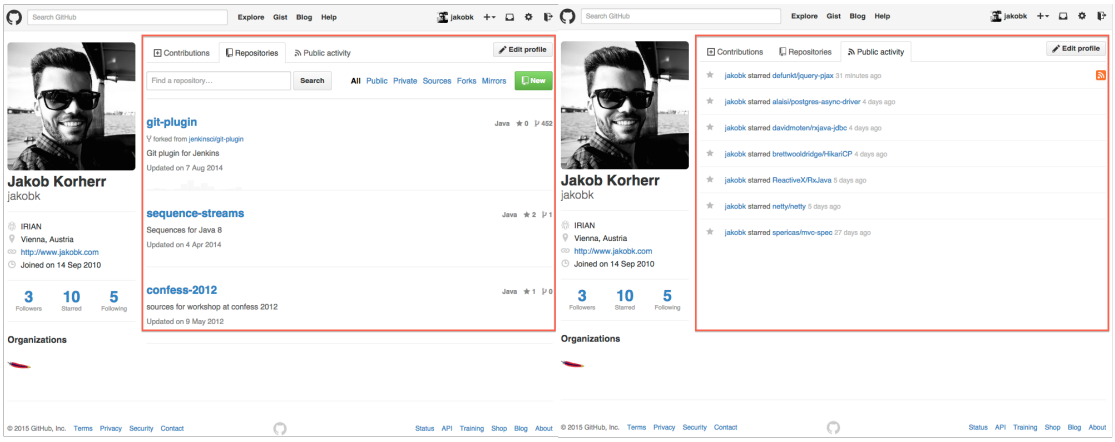


Figure 1.1: Two subsequent web pages from GitHub [55] with highlighted UI changes. The figure shows that only the content of the red box changes from the first request to the second one, whereas the other parts of the page stay the same.

The AJAX web application model, however, introduces a bookmarkability [87, 137, 138] problem, because the state of the web page changes in the browser after an AJAX request that modifies the browser DOM, but the location bar of the web browser does not reflect this new state [102, 143]. This problem has many consequences:

- The browser back- and forward-buttons do not work as expected [37, 107, 143].
- The current state of the web application is lost if the user refreshes the page [107, 143].
- Web crawlers cannot access all states of the web application [29, 56].
- The RESTfulness of the web application is lost [40].

To circumvent the bookmarkability problem, AJAX based web applications started using the *fragment identifier* to map the current URI in the location bar of the web browser to the current state of the application, because it is possible to change the current fragment identifier of a web page without having to reload the page in the browser [37,94]. The fragment identifier is defined in RFC 3986 Section 3.5 [16] as the component after a number sign ("#") character in an URI. It can be used to identify a secondary resource inside a primary resource [133]. Popular search engines also support crawling AJAX based web applications that use the fragment identifier for state mapping, if they use *hash-bang* fragment identifiers, i.e. fragment identifiers starting with #! [29, 56, 106]. This method of state mapping has been highly adopted, even on popular sites like Facebook [38] and Twitter [128, 135].

The usage of the fragment identifier, however, has a built-in flaw: It is not part of the request that is sent to the web server by the browser [26, 37]. This means that the web server cannot interpret the fragment identifier on the initial request, and therefore needs to send back a template page including a JavaScript handler to the browser. This JavaScript handler can then interpret the fragment identifier, and afterwards load the actual page content via AJAX in a second request

[67]. As a result, the browser needs more time to display the actual HTML page [37], and the user experience suffers [67]. Another problem is that there are now potentially multiple URIs pointing to the same resource [67], because the part of the URI before the fragment identifier only points to the entry page of the web application, and the fragment identifier alone determines the actual resource (i.e. the page content). Thus, if the web application has multiple entry pages, it also has multiple URIs to the same resource, which makes long time bookmarkability support especially hard [26].

To address these problems, the HTML5 specification introduces new functionality to the browser history API: the *pushState()* and *replaceState()* methods, and the *popstate* event [69]. Using these methods it is possible to manage the browser's history stack from JavaScript, and thus to change the URI in the browser location bar, without triggering a full page refresh [91]. In addition, a JavaScript handler can be installed to listen to browser history events triggered by the back- and forward-buttons of the browser [91].

Using the new HTML5 history API, it is possible to create a fully bookmarkable, single-page AJAX web application without using fragment identifiers. Popular websites like Google Plus [57], Facebook [38], Twitter [134], and GitHub [45] already utilize the HTML5 history API today. Moreover, novel JavaScript frameworks like AngularJS support this out of the box [6], in JavaEE [103], however, there is currently no built-in support for the history API.

Aside from these client side issues, most prevalent web frameworks have a scalability problem, because of the usage of [35, 136]:

- Stateful web servers, and
- Synchronous, blocking HTTP requests.

A great number of stateful web servers implement state management via HTTP sessions, which work by assigning a unique session identifier to the client browser and storing the actual client state on the server using this session identifier as the key [15]. On the one hand, this approach makes it easier for web application developers to build stateful applications on top of the stateless HTTP protocol, on the other hand this procedure leads to horizontal scalability problems, because the web application cannot simply be deployed on different server instances without adding server synchronisation and mechanisms for client routing and failover [42, 64]. Many existing Java EE web frameworks are built on top of the Servlet API [74], which has built-in support for HTTP sessions, and thus these frameworks also support this mechansism for storing state between HTTP requests. The Play Framework, which is not built on top of the Servlet API, introduced a different approach for handling client state without storing data on the server. The idea is to store the state in the client's web browser using an encrypted HTTP cookie. Altough this method restricts the maximum state size to about 4 kilobyte (i.e. the maximum size of a cookie), it allows the web application server to be completely stateless, which is a reasonable tradeoff [109]. This state management approach is not specific to the Play Framework, and can therefore easily be integrated in any Java EE web framework.

The other part of the scalability problem originates in the fact that traditional web frameworks use synchronous, blocking HTTP requests instead of asynchronous, non-blocking requests. This means that a client request consumes exactly one associated thread from a predefined thread pool on the web server, which is responsible for handling this request until it is

fully completed. Thereby this thread is blocked when waiting for the results of calls to external systems, e.g. a database server [42]. Again, on the one hand this approach simplifies web application development, because the synchronous processing model makes program logic easier to understand, implement, and maintain [132]. However, on the other hand this strategy consumes one thread for every concurrent client request, thus leading to a scalability problem, because the web server can only handle a maximum amount of concurrent threads, even if most of them are blocked and not doing any actual work [25]. When using asynchronous, non-blocking requests, the thread pool on the server can be reduced, because not every client request has exactly one associated thread [35]. That is because an incoming client request is handled by a thread from the server's thread pool only until a non-blocking call to an external system is performed. Instead of blocking the thread while waiting for the reply of the external system, the respective thread takes up other work, i.e. serve other clients. When the result of the external call is available, any free thread from the thread pool continues to process the original request. Starting with Servlet 3.0 [74], the Servlet API now also supports asynchronous request handling, i.e. it allows to suspend the processing of a request, which puts its associated thread back into the thread pool, and furthermore it allows to continue the suspended request on a different thread from the thread pool. Using this feature of the Servlet API, it is now possible to implement asynchronous, non-blocking request processing in Java EE, thus any framework built on top of Java EE can now easily adopt this way of request handling.

1.2 Problem statement

The problem, which this thesis aims to solve, is that there is currently no built-in support in Java EE for developing scalable single-page web applications, which use AJAX partial page rendering together with the HTML5 history API. Although it is, of course, possible to use the history API, partial page rendering, and asynchronous request processing in Java EE, there is currently no declarative framework support for it.

Addressing this problem involves solving the following sub problems:

- How to accomplish performant partial page evaluation and rendering of web pages in Java EE, and which server-side rendering technology should be used?
- How to declare those parts of the page, which should be re-rendered in a certain request?
- How to utilize the HTML5 history API to create a navigable browser history stack?
- How to map safe and unsafe interactions [139] to URIs on the history stack, i.e. how to handle form submissions and POST requests?
- How to implement bookmarkability, i.e. how to handle full page requests to any state of the web application?
- Which existing Java EE technologies can be utilized?

- How to allow web applications developed with this framework to scale, i.e.:
  - How can the server be made stateless in order to support horizontal scalability [64] without further considerations [42]?
  - How to support asynchronous, non-blocking requests [35] in order to serve a great number of concurrent requests on a single server, and can this be done in a declarative way, e.g. by using reactive extensions [112]?

1.3 Aim of the work

This thesis aims to create a novel web framework based on standard Java EE [103] technology that solves the above problems using a declarative approach. The framework should use server-side rendering, as opposed to client-side rendering, which is used in popular JavaScript single-page web application frameworks like AngularJS [2], EmberJS [30], React [111], Google Web Toolkit [58], and others.

Additionally, an example single-page web application, which is built using this framework, is developed and presented. This example application demonstrates the framework's capabilities on many common use cases of state of the art web application development. Concretely, it showcases:

- RESTful [40] navigation with partial page rendering, including bookmarkable URIs, and back- and forward-button support.
- Form submission and input validation.
- Authentication management (i.e. login of existing users, and sign up of new users).
- Asynchronous, non-blocking HTTP request handling [35].
- Horizontal scalability [64, 136] without server synchronization.

Based on this example web application, an evaluation of response times and response sizes is performed, comparing the partial page requests applied by this framework with traditional full page requests. In addition, a load test of the example web application is performed, which compares the response times of the application serving a great number of concurrent requests using traditional synchronous, blocking request processing on the one hand, and asynchronous, non-blocking request handling on the other hand.

This thesis does not aim to create a fully backward compatible framework, which also runs in browsers that do not implement the HTML5 history API. This means that the framework does not implement a fallback to URIs with fragment identifiers, if the HTML5 history API is not supported by the browser, but it rather fails to work in those browsers. This decision is supported by current browser usage statistics for March, 2015, which state an availability of the HTML5 history API for 88.5% of global internet users, 93.26% of internet users in Europe, and 94.39% of internet users in Austria [20].

Another non-target of this thesis is to provide a solution for pushing realtime data from the web server to the client browser, which can be implemented by utilizing WebSockets [39, 108], or SockJS [121], because this functionality imposes additional constraints on the server-side architecture when aiming for scalability. This topic is therefore left open for future work.

Any code produced during the development of this thesis is licensed under the Apache License, Version 2.0 [10].

1.4 Methodological approach

This thesis uses an iterative approach in order to develop the desired framework. In the first iteration, the basic functionality of the framework is developed in a prototype. This includes:

- Navigation and form submission using AJAX, by employing the Hijax approach [84].
- Utilization of the HTML5 history API for handling the browser history stack.
- Definition of the HTTP communication sub-protocol, needed for distinguishing between full page and partial page requests, and for carrying additional metadata.
- Selection of a server-side rendering technology.
- Implementation of partial page rendering using the chosen rendering technology.
- Definition of the syntax for defining the parts of the page, which should be re-rendered on the following request.
- Design and implementation of a basic Java EE adapter, including partial model evaluation, used for partial page rendering.
- Adoption of the state management approach implemented in the Play Framework, in order to create a real stateless server.

This prototype is then analyzed, and based on the findings, it is further improved. This includes:

- Refinement of the Java EE integration and usage of existing Java EE standards.
- Enhancement of the partial page rendering approach.
- Modularization of the framework's JavaScript using AMD (asynchronous module definition), and optimization for usage in a production environment [12, 90].
- Implementation of declarative support for asynchronous, non-blocking request handling by using reactive extensions [112].

During the design and development of the prototype, an example web application is created side by side, in order to test the functionality of the framework, as well as to reflect its suitability for state of the art web application development. In the end, the framework is evaluated using this example web application.

1.5 Structure of this thesis

This thesis is organized as follows:

- Chapter 2 explains the HTML5 history API in more detail, and it presents existing solutions for history management in popular single-page JavaScript web frameworks. Moreover, it outlines relevant Java web application frameworks and their support for asynchronous request processing. Finally, it gives an overview on related work, i.e. it references and explains various approaches that utilize the HTML5 history API in order to build bookmarkable single-page web applications, and it presents different approaches for building scalable web applications.

- In Chapter 3 the prototype for the web application framework based on Java EE is designed and implemented. This chapter therefore addresses the basic problems outlined in Section 1.2. The result of Chapter 3 is a working prototype of the framework, which can already be used to build bookmarkable single-page web applications on top of a stateless Java EE server.

- Chapter 4 analyzes the prototype created in Chapter 3, and elaborates the work on the prototype, in order to create a tighter integration with Java EE, and therefore a better developer experience. In addition, it makes the framework production ready for actual web application development by modularizing and optimizing the framework's JavaScript. Furthermore declarative support for asynchronous, non-blocking request processing is implemented using reactive extensions [112].

- Chapter 5 presents a sample single-page web application in order to exemplify the usage of the framework developed in Chapters 3 and 4. This chapter therefore aims as a proof of concept.

- The framework developed in this thesis is then evaluated in Chapter 6 on the basis of the example application presented in Chapter 5. The evaluation compares the amount of transferred data and the response times of the partial requests issued by the framework with traditional full page requests. Moreover, it also compares the response times of the web application serving a great number of concurrent requests when using asynchronous, non-blocking request handling as opposed to traditional synchronous, blocking request processing.

- Finally, Chapter 7 concludes the work of this thesis and gives an outlook on future improvements of the web application framework developed in this thesis.

CHAPTER 2

State of the art and related work

2.1 The HTML5 history API

The HTML5 specification adds two new methods to the browser's history interface [69]:

- *window.history.pushState(data, title [, url])*
- *window.history.replaceState(data, title [, url])*

With these two methods it is possible to push a new entry to the browser's history stack, and to replace the current entry of the browser's history stack, respectively. Additionally, the URI of the browser's location bar can be visibly changed by both methods, without having to reload the current page in the browser [91, 98]. This is not possible in previous versions of HTML, with the exception of the fragment identifier [37, 94].

In addition, the *PopStateEvent* is introduced, which can be listened to using the name *popstate* on the current window object [68]. With this event it is possible to listen to changes to the browser's history stack issued by the browser's back- and forward-buttons (or the related JavaScript functions).

Availability

The HTML5 history API is supported by almost all modern browsers. Table 2.1 gives an overview of supported browser versions.

Current browser usage statistics state an availability of the HTML5 history API for 88.5% of global internet users, 93.26% of internet users in Europe, and 94.39% of internet users in Austria for March, 2015 [20].

Browser	Versions
Chrome	5+
Firefox	4+
Internet Explorer	10+
Safari	6+
Opera	11.5+
iOS Safari	5.1+
Android Browser	4.3+

Table 2.1: Overview of popular web browsers supporting the HTML5 history API [20].

Example

An online example of the HTML5 history API can be found at [27]. This example shows a page embedding a photo gallery of different dogs. The gallery provides a *Previous* and a *Next* link to change to the previous or next photo, respectively. Figure 2.1 shows a screenshot of the page.

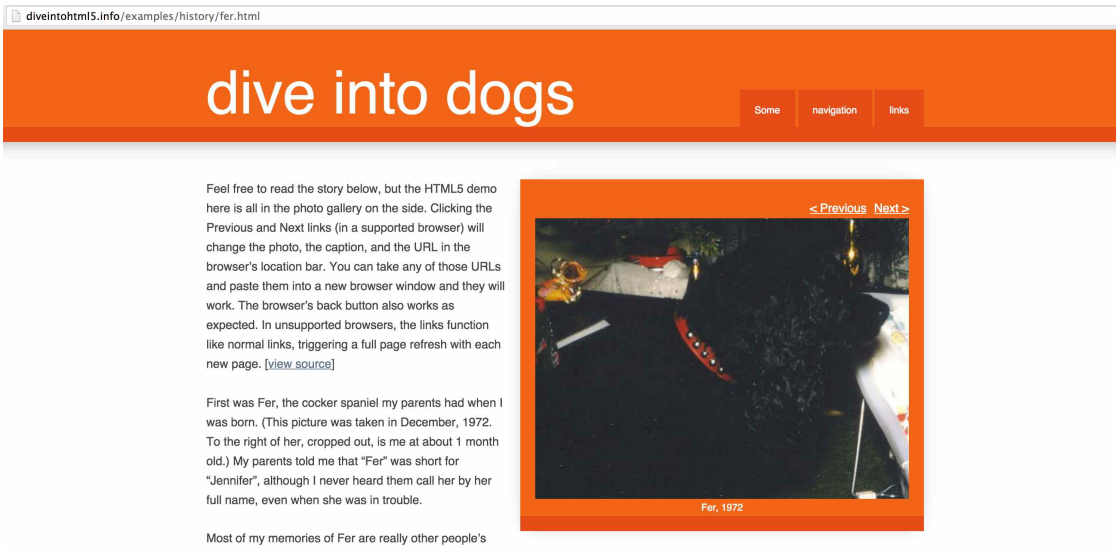


Figure 2.1: Screenshot of the initial page of the HTML5 history API example at [27].

The links for changing to the previous or next photo are thereby decorated with a JavaScript click handler, which fetches the gallery page of the requested dog via AJAX, exchanges the existing gallery contents with the updated ones in the browser DOM, and calls *history.pushState()* to change the current URI in the browser's location bar, instead of issueing a full page request to the next page. Using Chrome's developer tools, this behaviour can be observed. Figure 2.2 shows a screenshot of the same page after clicking on the *Next* link, with Chrome's developer tools enabled.

In Figure 2.2, it can be seen that the URI of the browser's location bar changed from *fer.html*

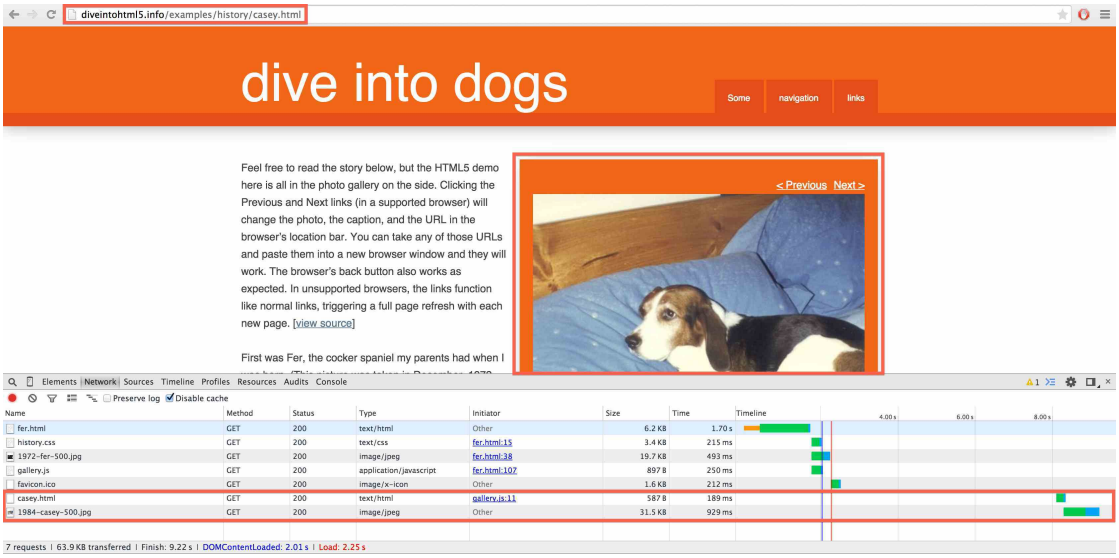


Figure 2.2: Screenshot of the same page shown in Figure 2.1 after clicking on the *Next* link.

to *casey.html*, and the content of the gallery changed to display the photo of the next dog. Chrome's developer tools, however, show that the page was not fully reloaded, but rather an AJAX request to *gallery/casey.html* and another request to receive the next image were executed.

Caveats

Although most modern browsers support the HTML5 history API as discussed above, not all of these browsers implement the history API in the same way, and unfortunately, some even introduced bugs into their implementation [66]. In order to circumvent these bugs and to ease the adoption of the HTML5 history API, the polyfill [113] called *history.js* [66] can be used. Using history.js as a wrapper of the plain HTML5 history API has the following benefits:

- The various different browser behaviours and browser bugs [66] can be ignored.
- If required, history.js can transparently fall back to *fragment identifiers* in browsers that do not support the HTML5 history API.

This means that in addition to the supported browsers listed in table 2.1, history.js also supports the browsers shown in table 2.2, however, falling back to fragment identifiers.

2.2 History management in single-page JavaScript web frameworks

Single-page JavaScript web frameworks execute most of the application logic and almost their entire rendering logic in JavaScript. The web server merely serves the template page including

Browser	Versions
Firefox	3
Internet Explorer	6, 7, 8, 9
Safari	4, 5.0
Opera	10, 11.0
iOS Safari	4.2, 4.1, 4.0, 3.2

Table 2.2: HTML4 web browsers supported by history.js using fragment identifiers [66].

the JavaScript sources, and later aims as the backend for the REST calls from the application's JavaScript and to reload additional resources like HTML templates, CSS files, more JavaScript modules, or images [41,94,106]. As an example, Figure 2.3 shows the structure of a single-page web application created using EmberJS.

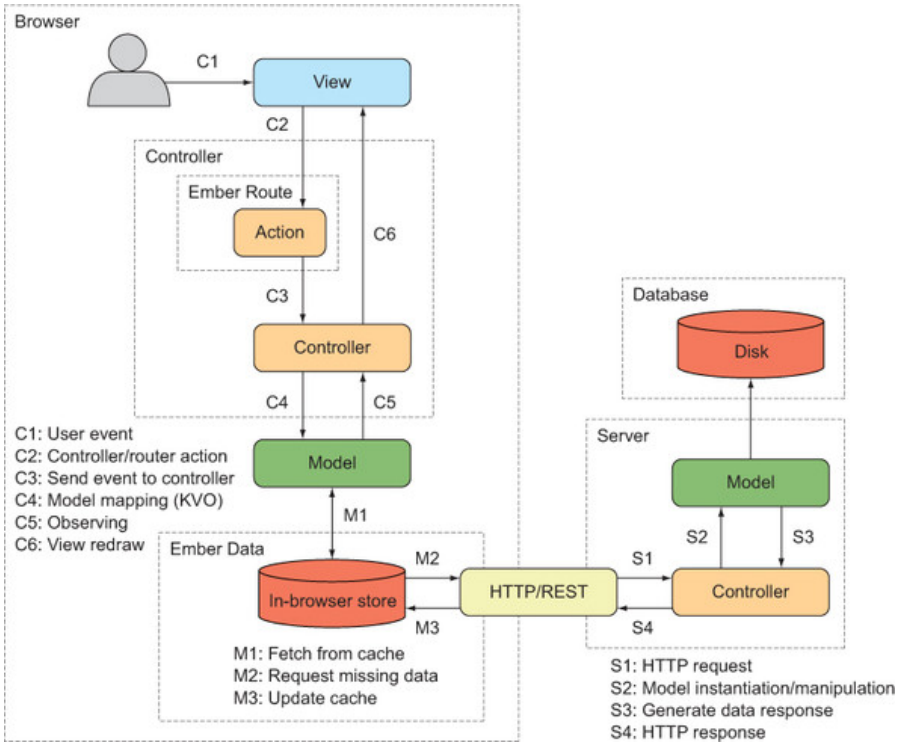


Figure 2.3: Typical web application structure in EmberJS [119].

This kind of web application frameworks have become very popular recently, because of, but not limited to, the following facts [41,90]:

- JavaScript execution speed increased dramatically in the last years [62].
- New JavaScript APIs were added in HTML5, e.g. *localStorage*, *WebSockets*, or *Web Workers*, which ease web application development in JavaScript.

- Server-side JavaScript environments, like *node.js*, allow the development of server-side and client-side code in JavaScript [85, 132].

- A big development ecosystem has evolved, including tools like *npm*, *bower*, *grunt*, or in-browser testing frameworks like *karma*, or *jasmine*.

- The introduction of languages that compile into JavaScript, e.g. *CoffeeScript*, or *Dart*.

As mentioned in the introduction of this thesis, many single-page JavaScript web frameworks already support history management via fragment identifiers or even via the HTML5 history API. The following subsections explain the history management of representative JavaScript web frameworks.

AngularJS

AngularJS [2] provides an abstraction over the browser's location bar and history stack: the *$location* service. This service provides a two way binding between the browser's location bar, and its internal state. External changes to the browser's location bar are therefore immediately reflected in the *$location* service, and changes to the *$location* service from the AngularJS application are instantly synchronized to the browser's location bar and history stack. In addition, it provides various methods to access the current protocol, host, port, path, search, and hash of the active URI [6, 61].

The *$location* service supports two configuration properties, which can be set using *$locationProvider* [3], as shown in Listing 2.1.

```
1 $locationProvider.html5Mode(true).hashPrefix('!');
```

Listing 2.1: Using *$locationProvider* to configure the *$location* service.

Per default, AngularJS uses fragment identifiers to map the current state of the application to the browser's location bar. In order to use *hash-bang* fragment identifiers [56], the *hashPrefix* property has to be set to *!* as shown above.

Setting the *html5Mode* property to true, however, enables the usage of the HTML5 history API in supported browsers, whereas in older browsers the service falls back to using fragment identifier URIs. Thus, using the HTML5 history API is transparent to the Angular application, because it is abstracted by the *$location* service. There is, however, some additional effort in order to fully support the HTML5 history API:

- The main HTML file of the application must contain a <*base*> tag, in order to correctly interpret relative URIs.

- The web application server must serve the main HTML file of the application for every valid URI that is created by the application, i.e. it must support URI rewriting. Otherwise the application would break if the user refreshes the page or uses deep linking [137] into the application.

The *$location* service can be used together with the *ngRoute* module, in order to allow declarative routing of the application's URIs to controllers and views (HTML partials) [4]. This mapping can be declared using *$routeProvider*, as shown in Listing 2.2.

```
$routeProvider.
  when('/', {
    controller: ListController,
    templateUrl: 'list.html'
  }).
  when('/view/:id', {
    controller: DetailController,
    templateUrl: 'detail.html'
  }).
  otherwise({
    redirectTo: '/'
  });
```

Listing 2.2: Declarative route configuration in AngularJS using *$routeProvider*.

All supported application URIs can be configured with respective *when* functions, and for the case that no defined URI matches, a default case can be defined using *otherwise*. A *when* function thereby relates a (possibly parameterized) application URI to a controller and a HTML view template. If the application URI changes, the *$route* service matches the new URI to the appropriate *when* case, add the defined HTML view template to the element with the *ng-view* directive in the browser DOM, and initialize and call the associated controller, who is then responsible for providing data for the template view [61].

EmberJS

EmberJS [30] is another popular single-page web application framework that is completely written in JavaScript, however, its state management approach is quite different from the one of AngularJS. Whereas AngularJS simply matches URIs to controllers and HTML view templates, EmberJS allows to create hierarchical structures of resources and routes [119], which can be defined using *App.Router.map()*, as shown in Listing 2.3.

```
App.Router.map(function() {
  this.resource('post', { path: '/post/:post_id' }, function() {
    this.route('edit');
    this.resource('comments', function() {
      this.route('new');
    });
  });
  this.route('catchall', {path: '/*wildcard'});
});
```

Listing 2.3: Declaration of resources and routes in EmberJS [34].

Resources are thereby understood as defined in the REST architectural style [40], i.e. *nouns* that relate to an actual object used in the respective controller, whereas *routes* are *adjectives* or *verbs* that express operations modifying a resource [34]. EmberJS uses naming conventions to look up the associated controller and route classes, as well as the appropriate view from

the resource and route structure. If any of these objects cannot be found, it is automatically be created by EmberJS. The mapping defined in Listing 2.3 results in the structure shown in table 2.3.

URI	Route Name	Controller	Template
/	index	IndexController	index
N/A	post	PostController	post
/post/:post_id	post.index	PostIndexController	post/index
/post/:post_id/edit	post.edit	PostEditController	post/edit
N/A	comments	CommentsController	comments
/post/:post_id/comments	comments.index	CommentsIndexController	comments/index
/post/:post_id/comments/new	comments.new	CommentsNewController	comments/new
any other URL	catchall	CatchallController	catchall

Table 2.3: Resulting application structure in EmberJS of mapping from Listing 2.3 [34].

The table shows that *resources* get an associated controller and view template, but they cannot be directly addressed with an URI. However, for every *resource* an *index route* and its associated controller and view template are automatically created. The last entry in the table applies to all unmatched URIs of the application, which allows to implement a behaviour like the *otherwise* mapping in AngularJS.

In EmberJS, every resource controller is responsible for setting up its respective state and providing it to its sub-controllers. The sub-controllers can then either use this state to setup their own state (resources), or they can alter this state (routes). In order for this to work correctly, EmberJS ensures that the controllers are inititalized in the correct order, as defined in the application's route mapping.

The URIs resulting from the configuration in *App.Router.map()* (e.g. as shown in table 2.3) are mapped to the browser's location bar exactly like in AngularJS: Per default, fragment identifiers are used, however, EmberJS can be configured to use the HTML5 history API instead [31]. Similarly, an application base path and URI rewriting on the server have to be configured too, if the HTML5 history API is used. The preferred method can be specified as shown in Listing 2.4.

```
1 App.Router.reopen({
2   location: 'history'
3 });
```

Listing 2.4: Configuring EmberJS to use the HTML5 history API instead of fragment identifiers.

In addition to *hash* (default) and *history*, EmberJS also supports *auto*, which utilizes the HTML5 history API, if available, and otherwise falls back to fragment identifiers [31], just like in AngularJS.

Google Web Toolkit (GWT)

The Google Web Toolkit is also a single-page JavaScript web framework, although a very special one, because the actual web application development is not done in JavaScript like in AngularJS

or in EmberJS, but rather in Java, which is then compiled to JavaScript by a special compiler [99, 129].

GWT has built-in support for history management via fragment identifiers [59], but it does not support the HTML5 history API out of the box, however, some external plugins add the missing functionality to GWT, e.g. the *gwt-pushstate* plugin on GitHub [47].

For history management, GWT provides two APIs:

- Basic history management using the class *com.google.gwt.user.client.History, misc:gwt-javadoc-history*.
- Higher level state management via *Places* and *Activities* [129].

The class *com.google.gwt.user.client.History* provides methods to navigate through the browser history, to access the current history token, to add a new token to the browser's history stack, or to replace the current token on the stack. In addition, a *ValueChangeHandler* can be registered to listen to changes of the browser's history stack [60]. This class is thus very similar to the browser's history interface in HTML5 [69], although it only supports to push and replace tokens, i.e. URI fragments, rather than full URIs. In Javascript, GWT uses different implementations of *com.google.gwt.user.client.History* for the various browsers in order to achieve this history mechanism. As of GWT 2.1, supporting browsers use HTML5's *onhashchange* event, IE6 and IE7 use an *iframe*, and all other browsers are based on a timer checking for fragment changes on the URL [129]. This means that for history support in Internet Explorer 6 and 7, the HTML in Listing 2.5 has to be added to the host page.

```
1 <iframe src="javascript:''"
2           id="__gwt_historyFrame"
3           style="position:absolute;width:0;height:0;border:0"></iframe>
```

Listing 2.5: The *iframe* needed to implement GWT's history mechanism in IE 6 and 7 [59].

Evidently, a basic history management can be implemented using *com.google.gwt.user.client.History*, however, due to the history tokens being only string values and GWT not including any parameter handling logic for history tokens, a custom parsing logic, especially for considering parameter values passed via the history tokens, must be implemented by the web application developers themselves.

For a more sophisticated history management, the *Place* API from GWT's MVP (model view presenter) approach [129] can be used. This API builds on top of the basic history mechanism, and provides an automatic mapping from history tokens to *Places*, including support for parameter passing. In this approach, the history tokens start with a place prefix that uniquely identifies the *Place* implementation, followed by a divider character, and finally a place-specific token, which is created and parsed via a place specific implementation of *PlaceTokenizer*. This tokenizer is responsible for serializing the state of a place into a string, and to create a place from a serialized string value. Here, the parsing logic for parameter values still has to be implemented by the web application developers, however, it is always specific to an associated *Place*. Thus, in contrast to having one big history token parsing logic, this approach allows to have many *Place* specific parsing logics, and in addition, parsing the *Place* identifier and finding the correct *PlaceTokenizer* is carried out by GWT.

All of the above functionality also works with the HTML5 history API, if the *gwt-pushstate* plugin from GitHub [47] is used, because this plugin only adds a different JavaScript history implementation to the application, while GWT's history API in Java remains unchanged. The usage of the HTML5 history API is thus transparent to the web application.

Conclusion

In conclusion, it is very easy to use the HTML5 history API with the above single-page JavaScript web frameworks, because no additional effort is needed from an application point of view, and only little work is needed on the server-side to provide the necessary URI rewriting. Deep-linking URIs can easily be supported in AngularJS by utilizing the *ngRoute* module, in EmberJS by defining the respective *resource* and *route* hierarchy, and in GWT by using the *Place* API. Nevertheless, due to these frameworks being client-side single-page web application frameworks, additional HTTP requests are always needed in order to process the initial request issued by a user. The first request loads the application's main HTML template containing the JavaScript module references, the next requests load all necessary application resources, and then after the application has been initialized in JavaScript, the following requests load the data required by the respective controller via REST (or a RPC mechanism). This means that the initial loading time is increased, compared to other web application frameworks that do not need to fetch additional data from the server via AJAX once the main web page is loaded.

Regarding scalability, this kind of frameworks rely on the server, which serves the AJAX requests issued by the frameworks' JavaScript in the client browser, in order for the respective web applications to scale. Accordingly, the frameworks themselves are not really concerned about scalability, because they make no assumptions about the server implementation. However, *node.js* is often used to implement the server backend for these kind of frameworks, and it uses a single-threaded asynchronous I/O eventing model [132]. In addition, most server APIs follow the REST architectural style [40], and are therefore stateless. Consequently, scaling single-page JavaScript web applications that use a RESTful API implemented by *node.js* is straightforward.

2.3 Java web frameworks with server-side rendering

This section gives an overview of popular Java web frameworks that use server-side HTML rendering, and explains their approaches to browser history management and the problems that come with it. Additionally, these frameworks are evaluated towards scalability.

Java Server Faces (JSF)

Java Server Faces is a standardized web framework that is part of Java EE [78]. It uses a component driven approach for building web applications, which means that the actual HTML that comprises the different web pages of the application is mostly abstracted away into UI components. In addition, the specifics of the HTTP communication are hidden by the framework. Thus, JSF's programming model resembles the one of Java Swing, even though JSF is used for building web applications. Although this approach can greatly ease web application development, these abstractions prohibit working closely with the underlying web model, which is why

companies like ThoughtWorks suggest to abandon JSF in favour of simpler frameworks that work directly with web technologies like HTTP, HTML, or CSS [131].

In the first versions of the specification, JSF makes heavy use of *POST* requests, even for simple navigation purposes. Unfortunately, this approach leads to serious problems when users start to use the browser's back button. Even worse, JSF's navigation mechanism has a built-in flaw, because upon navigation to a new URI, the browser's location bar displays the new URI only on the subsequent request, but not straightaway. Therefore, early JSF versions do not support bookmarkability. This behaviour is revised in newer versions of the specification by the introduction of *GET* support, stateless views, and the adoption of the *POST-REDIRECT-GET* mechansism. The latter uses HTTP 302 redirects for *POST* requests that perform a navigation in order to issue a *GET* request to the new URI. Using this new approach, basic bookmarkability support can be implemented, however, doubling the number of HTTP requests.

Starting with JSF 2.0 [73], the framework supports partial AJAX requests, which allow to execute and render only parts of the current page. Unfortunately, due to JSF's architecture, the framework still has to restore and process the whole page on the server, however, only the referenced parts of the partial request are really rendered. This means that the server has to perform nearly the same work for a partial request as it has to undergo for a full page request. Therefore, using partial AJAX requests hardly speeds up request processing in JSF. On top of this, JSF's partial rendering mechanism does not implement any history management, which breaks the framework's bookmarkability support.

Since JSF provides stateful components, it needs to persist this state between HTTP requests. This is accomplished by either storing the state on the server in the client's session, or by serializing it to a hidden input field in the HTML response, which is resubmitted to the server on the next request. Whereas the first approach is obviously stateful, the latter approach allows to build a stateless server with the cost of increased request and response sizes. However, JSF also promotes the usage HTTP sessions, which in turn makes the server stateful again.

Finally, JSF uses a synchronous processing model with no support for asynchronous request handling. Although newer version of JSF are built upon Servlet 3.0, which supports asynchronous request execution [74], JSF makes heavy use of contextual instances that are bound to the current thread, e.g. the *FacesContext*. Thus, introducing an asynchronouns processing model in JSF requires considerable effort.

JSR 371: Model-View-Controller (MVC)

This web framework, which is currently highly experimental, is a result of the Java EE 8 community survey performed in 2014 [80]. The outcome states that 61% of the respondents would like Java EE 8 to include another model view controller (MVC) web framework alongside Java Server Faces. Thus, the Java specification request 371 [77] aims to create a specification for an *action* based MVC framework, as opposed to the existing *component* based MVC approach implemented by JSF. The action based approach thereby provides more fine-grained control over the actual web technologies, compared to the component based approach of JSF, which hides these technologies behind abstractions [79].

MVC 1.0 is layered on top of JAX-RS, which is the Java EE API for RESTful web services [75], instead of building upon the Servlet API [71, 79]. In addition, the following well-known

Java EE frameworks are utilized and integrated into the framework [79]:

- Contexts and Dependency Injection for Java (CDI)
- Bean Validation

Layering an MVC framework on top of JAX-RS is not new, because this approach has already been applied in a simple web framework called *Htmleasy* [70] in 2009, which is built upon *RESTEasy*, the JAX-RS implementation of JBoss. Utilizing the existing JAX-RS standard allows web application frameworks like Htmleasy and JSR 371 MVC to be very lightweight, because the whole HTTP communication, including path matching and parameter parsing, is already performed by JAX-RS. Furthermore, this approach allows developers to use almost the same API for building MVC applications and RESTful web services. Listing 2.6 shows an example *Controller* in JSR 371 MVC.

```
1  @Path("hello")
2  @Controller
3  public class HelloController {
4
5     @Inject
6     private Models models;
7
8     @GET
9     @Path("{name}")
10    @View("hello.jsp")
11    public void view(@PathParam("name") String name) {
12       models.put("name", name);
13    }
14 }
```

Listing 2.6: An example *Controller* in JSR 371 MVC, employing request mapping features of JAX-RS [72].

As JSR 371 MVC uses an action based MVC approach, the URIs used by the framework reflect the application state more precisely than component based frameworks like JSF. However, the specification draft does not include partial page rendering yet, which means that full page requests have to be used. This results in the web framework being subject to the bookmarkability problems that come with using full page requests, e.g. having to use HTTP 302 redirects like JSF in order to create bookmarkable URIs, or allowing unsafe operations to be part of the browser history, which may result in unintended behaviour [94]. What is worse, the HTML5 history API cannot be used appropriately with full page requests.

By building upon JAX-RS, JSR 371 MVC can utilize the asynchronous request processing mechanism of JAX-RS without further considerations. Therefore, asynchronous, non-blocking request handling can be accomplished with this web application framework, however, without any special support by the framework. Aside from that, JSR 371 MVC lacks support of a state saving mechanism other than HTTP sessions, thus implementing a true stateless server is hard to realize.

Spring Web MVC

The Spring Framework supports creating MVC based web applications using the module Spring Web MVC, which has been introduced as early as 2004 [117, 125]. Conceptionally, Spring Web MVC provides the same functionality that JSR 371 MVC currently seeks to specify for Java EE, however, without relying on JAX-RS, because Spring Web MVC includes all the path mapping and HTTP mediation features of JAX-RS itself. In addition, Spring's own bean container is favoured over CDI, though bean validation is supported like in JSR 371 MVC.

The HTTP request and path mapping of the current version of Spring Web MVC is very similar to the one used in the latest JAX-RS specification, and thus in JSR 371 MVC. Both support specifying the HTTP method, request path including path parameters, and query and matrix parameters [75, 125]. Furthermore, the type of the request body can be specified to enable automatic parsing of the body into an instance of the respective type, and both frameworks also support various different return types, which are used to render the actual HTTP responses by the frameworks. For these return values, the JAX-RS specification allows any kind of response entities, for which respective *MessageBodyWriters* have been registered in order to transform them into an actual HTTP response. Spring Web MVC, however, automatically interprets special return value types that correlate with the MVC pattern, while still allowing so called *HttpMessageConverters*, which are equivalent to *MessageBodyWriters*. Table 2.4 describes all allowed return types in Spring Web MVC.

Return type	**Meaning**
ModelAndView	view to be rendered including model values
Model	model for rendering the implicitely determined view
Map	as above, but as Map instance
View	view to be rendered with implicitely determined model
String	logical name of the view to be rendered
void	response is completed, no further actions are necessary
HttpMessageConverter types	entity to be rendered by the associated converter
HttpEntity<?>	as above, but entity wrapped into HttpEntity for adding headers
HttpHeaders	HTTP headers to include in the response with no body
Callable<?>	produces the return value asynchronously by a Spring thread
DeferredResult<?>	as above, but produces the value in a thread of own choosing
ListenableFuture<?>	as DeferredResult, just a different wrapper type
CompletableFuture<?>	as DeferredResult, but standard type introduced in Java 8
Other types	single model value for the implicitely determined view

Table 2.4: Overview of allowed return types and their meaning in Spring Web MVC [125, 126].

As can be seen in table 2.4, Spring Web MVC supports the asynchronous request processing of Servlet 3.0 by using special return types, i.e. declaratively [123]. Returning a *Callable* has the effect that the calculation of the actual return value is moved from the thread of the servlet container to a thread of Spring, but this calculation may still block that thread. True asynchronous, non-blocking request processing can however be implemented by utilizing *DeferredResult*, *Lis-*

tenableFuture, or the new *java.util.concurrent.CompletableFuture* introduced in Java 8, because here the calculation of the actual return value is performed by threads not known to Spring MVC. Listing 2.7 shows the usage of *DeferredResult* in an example.

```
@RequestMapping("/quotes")
@ResponseBody
public DeferredResult<String> quotes() {
  DeferredResult<String> deferredResult = new DeferredResult<String>();
  // Add deferredResult to a Queue or a Map...
  return deferredResult;
}


// In some other thread...
deferredResult.setResult(data);
// Remove deferredResult from the Queue or Map
```

Listing 2.7: Asynchronous request handling in Spring MVC using *DeferredResult* [123].

Due to the fact that the request model of Spring Web MVC resembles the one used in JAX-RS and thus in JSR 371 MVC, Spring Web MVC is subject to the same problems regarding bookmarkability support and usage of the HTML5 history API. Moreover, HTTP sessions are also the only built-in state management mechanism, and thus using stateful servers is promoted.

Play Framework

The Play Framework differs significantly from the other Java web frameworks presented above, because it is not built upon any Java EE technology, e.g. the Servlet API, but it rather implements the whole HTTP request processing lifecycle itself. Aside from that it is technically a *Scala* based web framework, however, also providing a Java API [28]. Nevertheless, Play shares the action based MVC approach with Spring Web MVC and JSR 371 MVC, and therefore the structure of the HTTP processing mechanism is very similar in all three frameworks. One difference though is that the Play Framework uses a central file called *conf/routes* for mapping HTTP request paths to their associated controller methods, whereas the other two frameworks employ Java annotations. Listing 2.8 shows a sample *Controller* in Play containing one simple *Action* method, and Listing 2.9 shows a possible HTTP request path mapping in *conf/routes* for this *Action* method.

```
package controllers;

import play.*;
import play.mvc.*;

public class Application extends Controller {

    public static Result index(String name) {
        return ok("Hello " + name);
    }
}
```

Listing 2.8: A sample *Controller* in Play containing one simple *Action* method [110].

```
1 GET   /hello/:name          controllers.Application.index(name: String)
```

Listing 2.9: HTTP request path mapping for the *Action* of Listing 2.8.

As type safety is one of the basic principles of the Play Framework, this HTTP request path mapping is also verified by the Play compiler. Furthermore, Play supports type-safe references to an *Action* method either from another *Action* method or from a *View* template, without using the actual HTTP path. Therefore, *conf/routes* is the only resource in a Play web application that actually contains HTTP paths, which notably simplifies path management and refactoring. Additionally, this approach promotes bookmarkability support of the whole web application.

As mentioned in Section 1.1, the Play Framework implements its own state management mechanism on top of the stateless HTTP protocol by using encrypted HTTP cookies. In addition, a *Flash* scope is provided in order to preserve simple values from one HTTP request to the subsequent one, usually applied in HTTP redirects [109]. Other state saving mechanisms, including HTTP sessions, are not supported by Play, which makes every Play application server stateless by design, and thus horizontal scalability can be achieved without further considerations.

On top of this, the Play Framework internally handles every HTTP request in an asynchronous, non-blocking way. Therefore, *Actions* should complete as fast as possible, i.e. in an asynchronouns, non-blocking manner, which can be achieved by using *Promises* [28, 110]. The following built-in asynchronous APIs already return *Promises*:

- *play.libs.WS* for calling external web services.
- *play.libs.Akka* for communicating with Akka actors.

The result of an asynchronous API call that returns a *Promise* of some type can be transformed into a Play *Result* as shown in Listing 2.10.

```
1 public static Promise<Result> index() {
2     return WS.url(feedUrl).get().map(response ->
3         ok("Feed title: " + response.asJson().findPath("title").asText())
4     );
5 }
```

Listing 2.10: Transforming the result of an asynchronous web service call into a Play *Result* by using *map()* [110].

Unfortunately, application code is sometimes required to block the current thread, e.g. when performing a synchronous database call. For this scenario, Play advices the application developers to execute the blocking operations within a different execution context that provides the expected level of concurrency, or even better, to use *Akka actors*. By adhering to the programming model dictated by the Play Framework, the resulting web applications can serve a high number of concurrent requests on a single server [110].

Concerning bookmarkability, the points mentioned about Spring WEB MVC and JSR 371 MVC are mostly also true for the Play Framework. Although the internal request processing mechanism differs significantly from these other frameworks, Play's approach to request mapping is conceptually very similar. One advantage of the Play Framework certainly is the support

for type-safe navigation, which eliminates a vast cause for errors in development. Additionally, the *Flash* scope makes it easier to perform HTTP redirects after POST requests, i.e. *POST-REDIRECT-GET* like in JSF, which improves bookmarkability support in regard to unsafe operations. However, just like the other frameworks, the Play Framework is designed for full page requests, and thus the HTML5 history API cannot easily be integrated.

Conclusion

All of the above Java web frameworks do not integrate well with the HTML5 history API. Only one of them has built-in partial page rendering support, whereas the others rely on full page requests. JSF's endeavors regarding bookmarkability support are still too little in order to build real RESTful web applications, because JSF is in its nature a component based MVC framework, and thus it deliberately does not bother about the actual request path matching. The other three frameworks show much better bookmarkability support, because they are action based MVC frameworks, and thus they are designed to be concerned about the request path matching. However, these frameworks are also designed to use full page requests, which prohibits the usage of the HTML5 history API. Of course, it is possible to build custom solutions for partial page rendering that even employ the HTML5 history API, e.g. by using jQuery to issue JSON requests to the server and then render the respective data on the client, but this approach clashes with the server-side rendering model of JSR 371 MVC, Spring Web MVC, and the Play Framework. In addition, this approach is exactly the idea behind the single-page JavaScript web frameworks presented in the previous section, and thus makes the usage of the above Java server-side frameworks obsolete to some extend.

Real stateless servers, and thus horizontal scalability, can effortlessly be achieved with the Play Framework, but not by using any of the other frameworks, because they all promote the usage of HTTP sessions. Apart from that, asynchronous, non-blocking request processing can be implemented with JSR 371 MVC, Spring Web MVC, and the Play Framework, however, Play providing the best integration with this request handling approach, because it uses asynchronous request processing per default, and it has built-in APIs for performing non-blocking operations in the backend. In conclusion, from the above frameworks scalability can best be achieved with the Play Framework.

2.4 Related work

In his PhD thesis [94], Mesbah covers single-page AJAX applications thoroughly. He motivates the usage of single-page AJAX applications over traditional multi-page applications, and also mentions history management via URI fragment identifiers. In chapter 2 of Mesbah's thesis, which is also published separately [95], the *SPIAR* (Single Page Internet Application aRchitecture) architectural style for AJAX applications is defined based on the REST architectural style introduced by Fielding [40]. SPIAR deviates from REST in some points in order to fulfill the needs of component based AJAX applications, which are compared to Java Swing applications in the paper. Thus, SPIAR is by design not resource based, as opposed to the single-page AJAX application architecture presented in this thesis.

Mesbah further introduces an approach for migrating multi-page web applications to single-page AJAX interfaces using SPIAR in chapter 3 of his PhD thesis, which is also published separately [96]. He presents a complete migration process, which extracts the navigational model from the multi-page web application, classifies the different pages based on their similarity, and finally results in a single-page AJAX application based on SPIAR. Moreover, a tool called *RETJAX* is developed, which implements the aforementioned approach. The tool is further used on an example multi-page web application for evaluation. Although it is important to define a migration process for existing web applications to single-page interfaces, this thesis focuses on developing new web applications based on the presented framework, rather than migrating existing ones.

The remainder of Mesbah's PhD thesis covers performance testing of AJAX data delivery techniques, crawling of AJAX based web applications, and automatic testing of AJAX user interfaces, which are all not handled in this paper.

Other works also describe ways for migrating traditional multi-page web applications to single-page AJAX interfaces. Keith [83, 84] presents the *Hijax* approach for progressively enhancing existing multi-page web applications with AJAX by intercepting HTML links and form submissions with JavaScript, and sending the data via AJAX instead. Unfortunately, Keith does not cover URI history management, i.e. using URI fragment identifiers or the HTML5 history API.

Oh et al. [102] use the idea of *Hijax* for transforming template based web applications into single-page applications that employ partial page rendering using AJAX. They transform existing JSP based applications, which use a common template JSP file on the server that includes the actual content of the different web pages from the respective content-only JSP files on full page requests. As defined in *Hijax*, they intercept the links and form submissions using JavaScript and send the associated data via AJAX instead. The server then processes only the requested content-only JSP file without processing the template JSP. On top of this, they use the HTML5 history API for history management and bookmarkability of the AJAX application. Moreover, an Eclipse plugin is presented, which enables an automatic migration process. Finally, the authors evaluate their approach by comparing it to *PJAX* [52], which is a jQuery plugin for combining AJAX partial page updates with the HTML5 history API. In essence, their approach is perhaps the closest concept to the one presented in this thesis, however, Oh et al. only support one common part that stays the same over subsequent AJAX requests, i.e. the template JSP, whereas all other parts necessarily need to be re-rendered, even if some of them are not changed. As a consequence, their technique does not include partial controller processing on the server, which is however part of this thesis.

Wang et al. [140] describe a similar, but less comprehensive transformation process using an automatic converting tool called *ACT*. Again, the authors concentrate on JSP based web applications. The core statement of their paper is that the automatic transformation process saves a lot of time compared to a manual transformation.

Apart from migrating traditional multi-page web applications to single-page AJAX interfaces, various works describe ways to refactor existing web applications in order to employ AJAX. Chu and Dean [24] use the TXL Source Transformation Language to refactor JSP based web applications that show pageable data lists in order to allow AJAX based paging of the data

lists, without reloading the whole page. After some manual tagging of the application parts associated with list processing, their transformation tool generates a web service on the server supplying the necessary list data in XML, and the client JavaScript that calls the respective web service and transforms the received XML data into HTML, which is then inserted into the browser DOM. They do, however, not merge existing JSP pages, nor use any kind of history management mechanism.

Further works are concerned with AJAX enabled form submission. Ying and Miller [142] introduce a refactoring system called Form Transformation Tool (FTT) in order to support developers in implementing AJAX enabled form submissions. Their system does not only migrate the whole server communication to AJAX, it also generates client and server side code to enable input validation on both sides. However, the system only supports JSP based applications which use MySQL as the database. Another work in this area uses AJAX to prevent unintentional repeated form submissions [36], which occur when users use either the back or the refresh button of the browser on POST requests.

The benefits of using AJAX partial page rendering over traditional full page rendering are demonstrated by Ayuba et al. [14] by applying the AJAX technique in the development of a web-based catalog system, and evaluating response times and server load compared to full page requests. Their results show an average response time of 417 ms for full page requests, and 16 ms for partial page requests.

Mazzetti et al. [92] show that building an AJAX application that is subject to the REST architectural style is possible, by introducing a REST conform AJAX framework called *MIRAJ* and developing a web application using this framework.

Single-page user interfaces are endorsed by Willemsen [82] by comparing the workflow of online travel booking sites when using a single-page interface as opposed to traditional multi-page interfaces.

Han et al. [63] show how a web page can be split up into separate, self-contained parts, called *Pagelets*, which can be either static, or dynamic by utilizing AJAX. Their paper also evaluates client-side rendering as opposed to server-side rendering, and they compare Pagelets to other JavaScript based rendering engines, e.g. *Mustache*, *Closure*, or *Dust.js*.

However, splitting a whole web page into separate parts, which can be updated independently is not new. This technique has been introduced even before partial page rendering using AJAX has been added to web browsers, in order to save server resources. Several papers [17, 19, 21] from 2004 and 2005 introduce *fragments*, which are cached on the edge servers of a web application, in order to prevent the actual web application from recalculating those parts of the web page that are not changed, and therefore to improve the performance of the system. Since AJAX can not be used, those systems have to use full page requests between web browser and edge server, however, the edge server then communicates with the actual web application server in a partial page manner. Fragments can thereby by organzied hierarchically, allowing to cache as much as possible on the edge servers. Altogether, the papers show considerable improvements in response times and server loads.

In his Master's thesis [35], Erb explores different server architectures for building concurrent, scalable web applications. The prevalent synchronous, blocking model which uses one process or thread per client is compared to several event-driven, non-blocking approaches. More-

over, various concurrency concepts for applications and business logic are examined, including *thread-based*, *actor-based*, and *event-driven* concurrency models. Additionally, commonly used server side system architectures for building scalable web-based systems are presented, from load-balanced multi-tier architectures through to Cloud Computing.

Scalable web infrastructures are further explored by various papers. Li and Yang [88] present a solution for handling stateful HTTP session data in a distributed environment composed of multiple web application servers. Further works explore session management in scalable cloud infrastructures [118, 127]. Moreover, Vaquero et al. [136] and Chieu et al. [22] present methods for dynamically scaling applications in a cloud environment.

Finally, Souders [122] summarizes different methods for improving web application load times in order to improve user experience.

Conclusion and differentiation

Many existing works are concerned with the automatic refactoring of a traditional, JSP-based web application, which uses full page requests, to an AJAX-enabled one, which handles particular cases via AJAX instead of utilizing full page requests. The extent to which AJAX is utilized varies heavily in the above papers. Some use AJAX only for specific use cases, e.g. for displaying paginated data lists, or for handling form submissions, while others integrate AJAX tightly into the application. Certain approaches exclusively use AJAX for client-server communication, with the initial request from client to server being the only exception. These approaches therefore promote single-page web interfaces. Unfortunately, few papers discuss the utilization of the HTML5 history API, or at least a fragment based history management approach, for achieving state management and bookmarkability in pure AJAX based web applications. In contrast to the presented work, this thesis aims to create a novel web application framework that is not concerned with the migration of existing web applications, but rather addresses the development of new web applications. Thus, this thesis does neither present an approach that is based on JSP, which would allow a simpler migration process of existing JSP-based web applications, nor does this thesis present a migration process or even develop a tool for automatic migration.

The framework presented in this thesis further differentiates from the related work discussed above by employing a resource based approach, as opposed to a component based approach suggested by the *SPIAR* architectural style. Therefore, the framework does not use components as the smallest unit of partial processing, but it rather separates each page of the web application into several self-contained parts, which can be processed and re-rendered independently of each other. In order to accomplish this, the page templates (i.e. the view) must be split into separate parts, and in addition, the controller logic and the model must be divided to fit the separation of the web pages. This fragmentation of model, view, and controller is presented in some of the above papers that utilize AJAX, however, most of them use a less comprehensive approach, which only allows to separate the pages into the main page template and the actual page content. Nevertheless, some works actually do present a more fine-grained approach for page fragmentation, however most of them apply the separation inside the web server infrastructure and still use full page requests between client and server. In contrast, the framework developed in this thesis does not enforce any limit on the extent of page (content) separation. In addition, all page

fragments are recognized on the client, which allows compartmentalized partial-page processing between client and server using AJAX.

Finally, none of the above approaches for utilizing AJAX in web applications supports the usage of asynchronous, non-blocking request processing on the server, which is, however, a crucial part of this thesis. Furthermore, all of the above works rely on HTTP sessions for state management. Some papers discuss the handling of session data in a distributed server environment, but none actually emphasizes the use of a different approach for client state management, as presented in this thesis.

CHAPTER 3

Design and implementation of a prototype

This chapter develops the prototype of the web application framework *Mascherl*.

3.1 Fundamental request handling of the framework

In this section the basic functions of the framework are presented using a sketched request sequence of four web pages. The sequence of pages is shown in Figure 3.1.

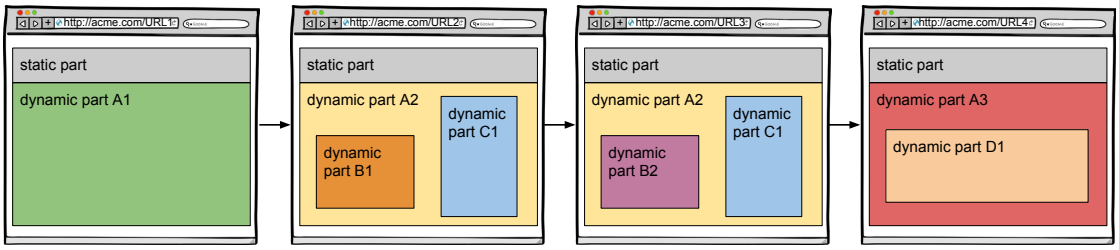


Figure 3.1: Sketched sequence of web pages, highlighting the page fragments that change between requests (in the first request A1 changes to A2, in the second request B1 to B2, and in the last request A2 to A3).

As can be identified in Figure 3.1, all web pages share a static part in the top (grey). The first page (*http://acme.com/URL1*) further includes one dynamic container A, with the associated content A1 (green). The second page (*http://acme.com/URL2*) also contains the dynamic container A, but with the content A2 (yellow), which itself consists of two dynamic containers B and C with the contents B1 (orange) and C1 (blue), respectively. Page three (*http://acme.com/URL3*) is very similar to the previous one, however, container B renders content B2 (purple) instead of

B1. Finally, page four (*http://acme.com/URL4*) again contains container A, but now with content A3 (red), which itself encloses container D with content D1 (light orange).

Now, in order to see the first page of this example, a user must visit *http://acme.com/URL1* in the web browser. For that the browser sends a standard full page HTTP request to the above URI, and the web server answers with a standard HTTP request that contains the full HTML of the requested page, including references to JavaScript and CSS files, which are then automatically fetched by the web browser. Figure 3.2 shows this process schematically.

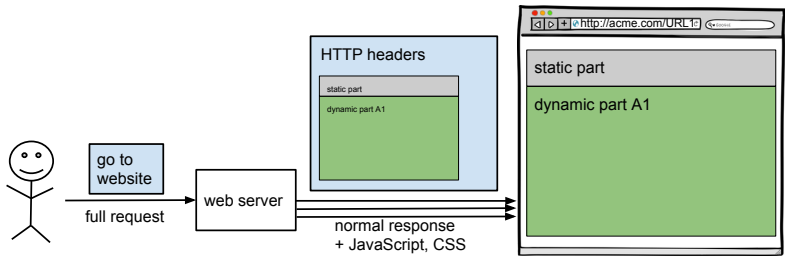


Figure 3.2: Schematic full page request processing for the first web page of Figure 3.1, showing that the full page is loaded, including all JavaScript and CSS files.

After the initial web page is fully loaded in the web browser, the JavaScript handler of *Mascherl* is called. This handler adds click handlers to all links of the page, as well as submission handlers to all forms of the page. These handlers are used to intercept potential full page requests issued by the browser, e.g. if a user clicks on a link on the page. Instead of letting the browser issue a full page request, the associated JavaScript handler changes the URI on the browser location bar to the new URI using the HTML5 history API, and performs an AJAX call to the server, specifying the requested target URI and the container of the new page, which should be rendered.

The web server answers this AJAX call with a HTTP response that contains only the requested container's HTML output, plus meta information like the new title of the web page. The AJAX response handler of *Mascherl* then parses the response, processes the given metadata, and finally replaces the content of the associated container in the DOM with the new HTML, while also adding the aforementioned link and form handlers to the new markup. Section 3.3 explains this process in more detail.

This behaviour, which is compliant to the *Hijax* approach introduced by Keith [83, 84], makes the user think that he or she navigates to a completely new page, although technically only one fragment of the page really is changed. Furthermore, the server only processes the associated container and not the whole page, and also only the new HTML of the container is transferred from the server to the web browser. Figures 3.3 and 3.4 visualize this mechanism, showing a navigation from web page one to two, and from web page two to three of the above example, respectively. The figures further illustrate that the sizes of the partial responses are smaller compared to the size of the full page request of Figure 3.2.

This Hijax based navigation between web pages however only works as long as the user uses controls of the web page (links, buttons, form submissions) for triggering the navigation, because then the JavaScript handler of *Mascherl* is able to intercept the default behaviour of the browser,

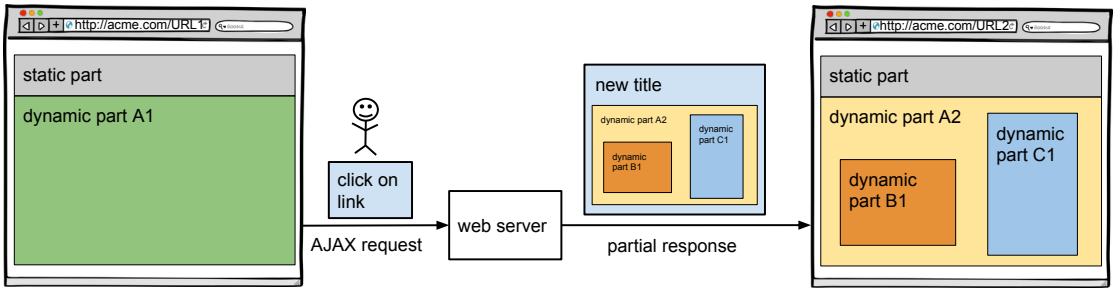


Figure 3.3: Schematic AJAX request processing in the first request of Figure 3.1, showing that only the contents of page fragment A2 and the new title of the page are loaded from the server.

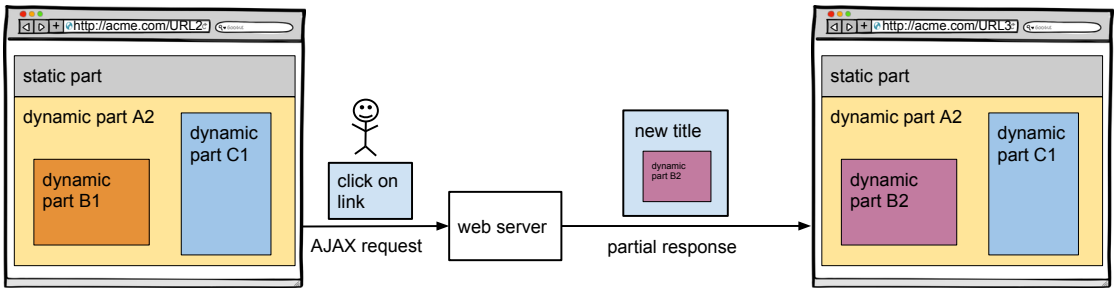


Figure 3.4: Schematic AJAX request processing in the second request of Figure 3.1, showing that only the contents of the small page fragment B2 and the new title of the page are loaded from the server.

i.e. sending full page requests. If the user clicks on the refresh button in the web browser, a new browser-triggered full page request is inevitable. The browser issues this new full page request to the current URI of the browser location bar, which has previously been changed (before the last AJAX request has been sent) using the HTML5 history API to reflect the current state of the web application. In this scenario, the web server has no other choice than to re-render the whole web page, however, this full page response exactly reflects the previous web page, which has been shown before the user triggered the browser refresh. Thus, the web browser shows the refreshed content of the whole page with only one full page request. Figure 3.5 shows this process schematically.

After the browser refresh is complete, the same JavaScript handler that processes the page's links and forms is triggered again, just as described before for the initial request to the web application. Therefore, if the user now once more clicks on a link, the same partial page processing is triggered again, just like before the full page refresh has been performed. The navigation from the refreshed page to the last page of the example is shown in Figure 3.6.

Apart from the browser refresh button, the framework also needs to handle the browser back and forward buttons appropriately. This is accomplished using the *PopStateEvent* of the HTML5 history API as described in Chapter 2. During the initialization process that is triggered after a full page request, the framework automatically adds a JavaScript handler that listens to

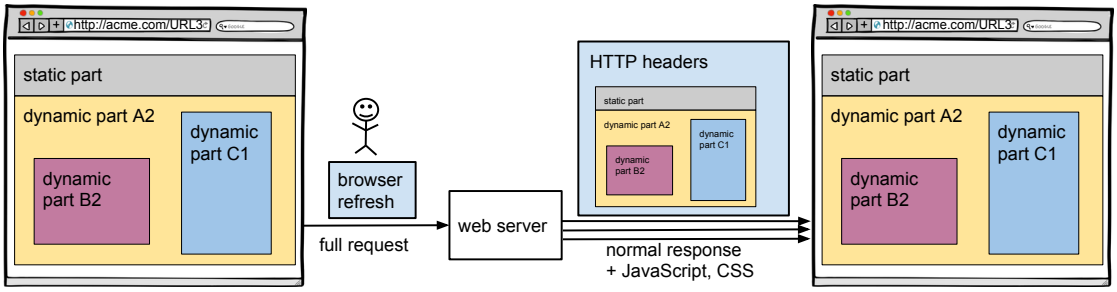


Figure 3.5: Schematic full page request processing if the user clicks on the refresh button in the browser on the third web page of Figure 3.1, showing that the full page including all JavaScript and CSS is loaded again.

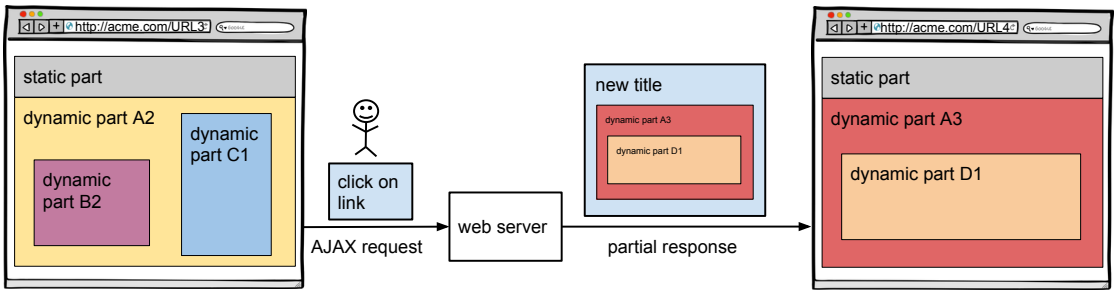


Figure 3.6: Schematic AJAX request processing in the last request of Figure 3.1, showing that only the contents of page fragment A3 and the new title of the page are loaded from the server.

the *PopStateEvent* in the web browser. This handler implements behaviour equivalent to the handlers that are added to the page's links and forms, i.e. it performs a partial page request via AJAX to the URI referred to in the *PopStateEvent*. The web server then processes and renders the outermost container of the current web page that needs to be changed in order to fully display the desired page reflected by the browser history entry. Figure 3.7 shows this process, triggered by a click on a browser back button on page four of the example page sequence.

This example illustrates that the framework is able to consistently handle all web page navigation eventualities (normal navigation via links, form submissions, browser refreshes, and back and forward navigation via the browser back and forward buttons, respectively) with partial page processing in a single-page interface.

3.2 Partitioning of page and model into fragments using containers

In order for the above request handling mechanism to work, all web pages of the web application and their associated models and model calculations need to be split into separate, independent parts, called *fragments*, and their associated containers that include the fragments. The framework needs to be able to process each container of every page independently from all other

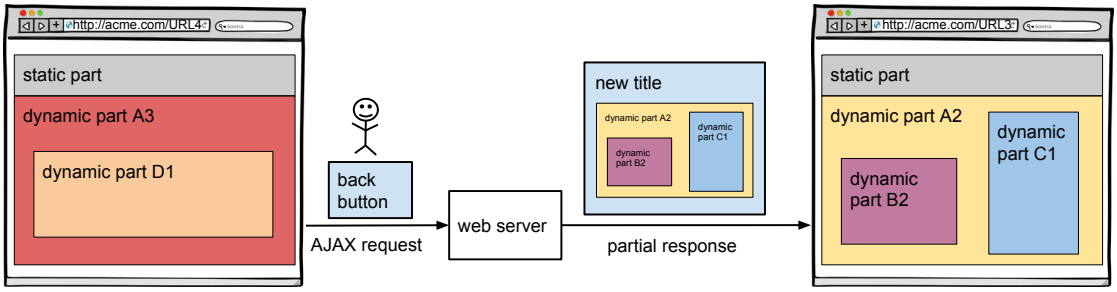


Figure 3.7: Schematic AJAX request processing from the last web page of Figure 3.1 to the previous one, issued by a click on the browser back button. The figure shows that even in this scenario only the contents of page fragment A2 and the new title of the page are loaded from the server.

containers of the web application, because each container may need to be processed at any point in time, if the respective partial page request is issued to the server.

In general, the framework can not anticipate which containers are processed together, with the exception of fragments that contain containers, which itself again include other fragments. If the container of such a parent fragment is processed, the fragment's containers and therefore its child fragments are also processed, however, the opposite is not true. A child fragment has no way of knowing that the container of its parent fragment is also processed in the current request. That is why every child fragment must be completely independent of its parent fragment, i.e. a child fragment must not rely on (model) data provided by a parent fragment.

Each page fragment consists of:

- A view, i.e. a template that eventually produces the HTML for the fragment, given an appropriate model. This HTML is later added to the associated container.
- A model that contains the values to render to above view.
- A self-contained model evaluation procedure that produces the above model.

The idea of splitting a web page into separate independent fragments is nothing new. Figure 3.8 shows an example taken from Challenger et al. [21].

To what degree a particular web page needs to be split into fragments depends heavily on the page layout, the underlying data dependencies, and the web application page flow. However, in *Mascherl* every web page needs to have an outermost *main* container, which contains the main fragment of the current page. This main fragment, which can, of course, contain further (child) containers, is the outermost dynamic unit of the web application. No partial page request can render markup outside of the main container.

To include the main container on full page requests, *Mascherl* uses a single HTML page, which is completely static. This page defines the common HTML structure of *all* web pages of the entire web application, and therefore declares the typical HTML <*head*> and <*body*> sections. Because this static page is shared by all pages, it needs to include all necessary JavaScript

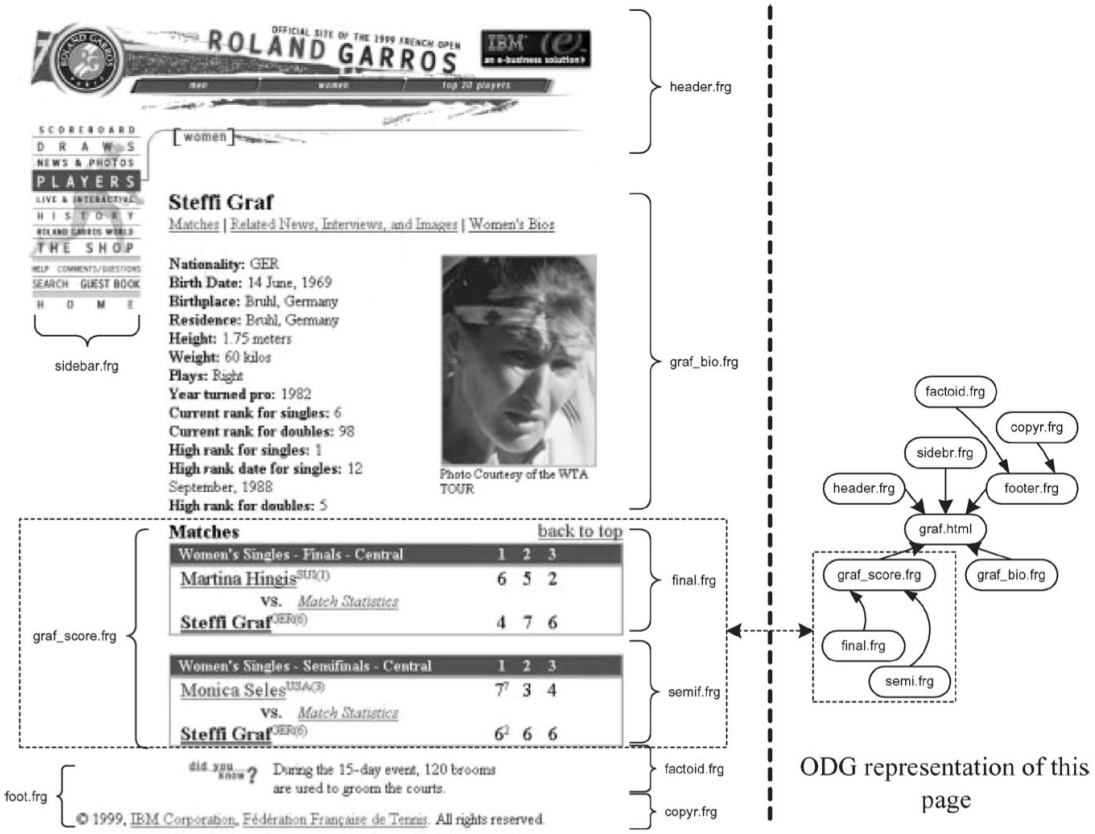


Figure 3.8: Sample screenshot showing the separation of a web page into fragments, taken from Challenger et al. [21]. The figure further shows the object dependence graph (ODG) of the web page's fragments.

and CSS files for the whole application. Finally and most importantly, this static HTML page is required to include the main container in the HTML <*body*> section. In a *Mascherl* based web application, this single HTML page needs to be placed into the web application root directory with the name *index.html*. Listing 3.1 shows an example of such a static HTML page.

```
1  <!DOCTYPE HTML>
2  <html>
3  <head>
4      <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
5      <title>{{title}}</title>
6      <link rel="stylesheet"
7            href="/webjars/bootstrap/3.3.4/css/bootstrap.min.css" />
8
9      <script src="/webjars/jquery/2.1.3/jquery.min.js"></script>
10     <script src="/webjars/bootstrap/3.3.4/js/bootstrap.min.js"></script>
11     <script src="/webjars/historyjs/1.8.0/scripts/bundled/html5/jquery.
           history.js"></script>
```

```
12      <script>
13          window.mascherl = window.mascherl || {};
14          window.mascherl.applicationVersion = "{{applicationVersion}}";
15      </script>
16
17      <script src="/mascherl/1.0.0/js/mascherl.js"></script>
18
19      <script id="error" type="text/x-error-template">
20          <p>Something went wrong... Please reload the page and try again.</p>
21      </script>
22  </head>
23  <body>
24  <h1>Mascherl Example</h1>
25
26  <div>Static content</div>
27
28  {{{@main}}}
29
30  </body>
31  </html>
```

Listing 3.1: An example of the main static HTML page (*index.html*) in a *Mascherl* based web application, which defines the basic HTML structure shared by all pages of the entire web application, and which defines the main container that includes the actual dynamic content.

The main container is thereby specified by `{{{@main}}}`, which is a template specific syntax that is further explained in Section 3.6. Listing 3.1 also shows that the HTML page is not entirely static, because it includes placeholders for the title of the web page and the application version.

The usage of this static HTML template for handling full page requests and the implementation of the inclusion and evaluation of page fragments is the subject of subsequent sections.

3.3 Using Hijax and the HTML5 history API on the client

As already mentioned, *Mascherl* implements the *Hijax* approach introduced by Keith [83, 84] in order to transform an otherwise multi-page web application into a single-page application. Furthermore, it has been stated that the HTML5 history API is used for adjusting the current URI of the browser location bar, and for handling the browser back and forward buttons. This section now covers the implementation of this process in JavaScript.

The JavaScript library

The JavaScript library of *Mascherl* internally uses *jQuery* to ease JavaScript development. Furthermore, *history.js* [66] is used rather then utilizing the HTML5 history API directly, in order to avoid various browser quirks.

After a full page request to the web server, the JavaScript library of *Mascherl* is initialized for the web page. The initialization procedure performs the following tasks, which are described in detail afterwards:

- Add a click handler to all links of the web page, which are not marked with *m-ignore*.
- Add a submission handler to all forms of the current page, which are not marked with *m-ignore*.
- Register a listener for the *statechange* events produced by *history.js*.

Handling of link clicks

As dictated by *Hijax*, all links on the web page are decorated with JavaScript handlers in order to prevent the browser from sending a full page request, and to send an AJAX request instead. In *Mascherl*, the associated JavaScript click handler is added to all links in the HTML <*body*> of the web page, with the exception of links specifying the *m-ignore* attribute. Links that should be marked with *m-ignore* by the application developer are links that target external web sites, because here a full page request is wanted.

The added click handler performs various tasks in order to accomplish the desired partial page update of the current page, which effectively transforms a multi-page application into a single-page one. To understand the behaviour of the click handler, Figure 3.9 first shows how the browser handles a link click, if the click handler of *Mascherl* is not installed.

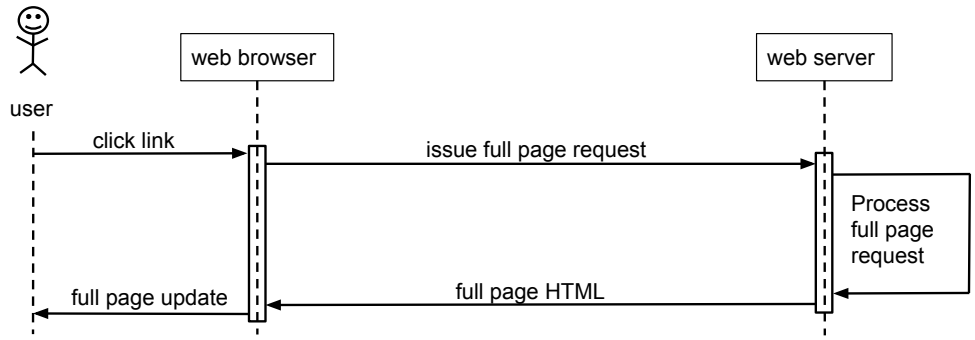


Figure 3.9: Sequence diagram of a normal full page request triggered by a link click.

This traditional browser behaviour is very simple: The web browser issues a full page request to the URI specified in the link's *href* attribute, the web server processes this full page request and sends the entire HTML of the requested page back to the web browser, and finally, the web browser displays the new HTML page. Employing *Hijax*, this simple behaviour becomes more complex, as shown in Figure 3.10.

Instead of letting the browser issue a full page request, the click handler first pushes the new URI specified by the link's *href* attribute to the browser history stack using *history.js*, then the click handler issues an AJAX GET request to the same URI, however including additional metadata in order to enable partial page processing on the web server. This additional metadata includes the name of the container that should be processed on the server, which is derived from the link's *m-container* attribute. Finally, the click handler prevents the default behaviour of the browser, i.e. sending the full page request. The web server now processes the partial

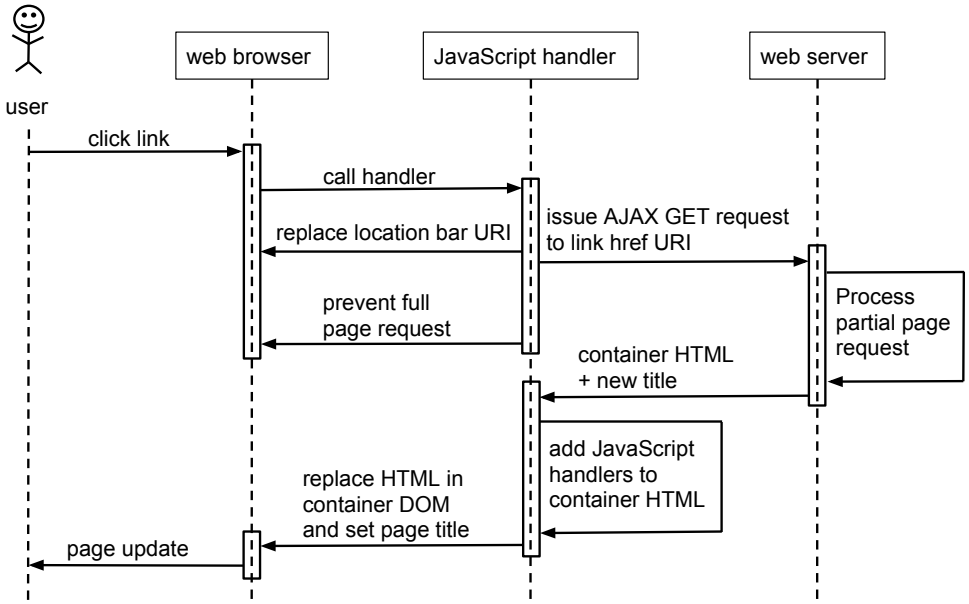


Figure 3.10: Sequence diagram of the process triggered by a link click in a *Mascherl* based web application.

page request and sends back a partial response to the web browser, which calls the associated AJAX response handler. This response handler adds *Mascherl's* JavaScript click and submission handlers to the response HTML, and finally replaces the content of the referenced container in the browser DOM with the new HTML markup from the partial response. Additionally, the title of the web page is changed using JavaScript to the new title received with the partial response.

Upon completion of this process, the browser DOM tree is effectively equal to the DOM tree resulting from the traditional full page request processing presented above, although actually only a partial page request has been performed.

Handling of form submissions

The standard handling of form submissions by the web browser is very similar to the traditional link processing behaviour illustrated in Figure 3.9, however, usually applying a HTTP POST request instead of a GET request.

The JavaScript handler of *Mascherl*, which intercepts form submissions, also works in a similar fashion as the JavaScript click handler presented in the previous section, but there are some subtle differences in the exact behaviour, as shown in Figure 3.11.

First of all, the JavaScript handler does not push a new URI to the browser history stack right away, but it rather only sends an AJAX POST request to the URI specified in the form's *action* attribute. In addition, it also prevents the default behaviour of the browser, i.e. triggering a full page submission. The web server now also processes the partial page request, and sends the rendered HTML fragment including the new title of the web page back to the browser. However,

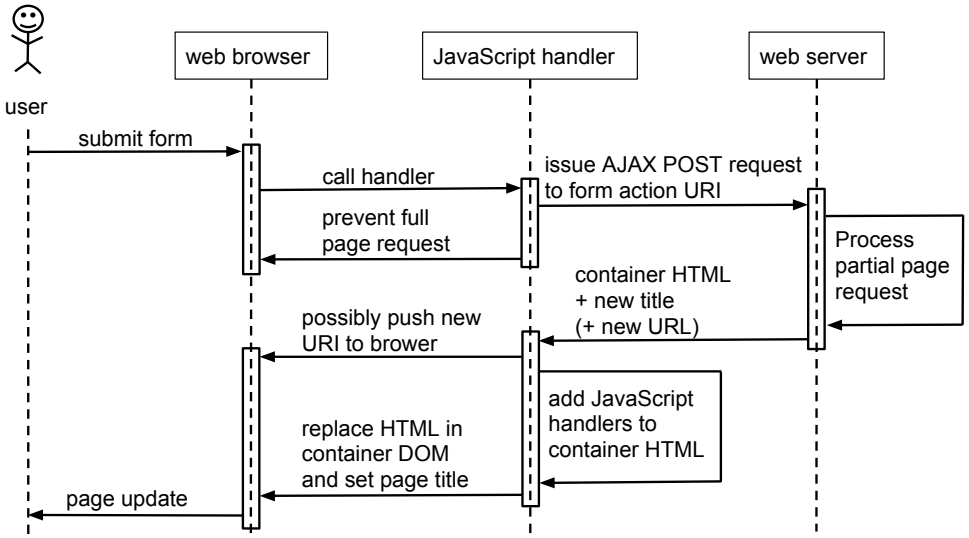


Figure 3.11: Sequence diagram of the process triggered by a form submission in a *Mascherl* based web application.

additionally the web server can also send back a new URI, which is then pushed to the browser history stack using *history.js* by the AJAX response handler in the web browser. Finally, exactly like in the link handling scenario, this response handler also adds click and submission handlers to the new HTML, replaces the HTML in the DOM, and sets the new title of the web page.

The reason why the behaviour of the form submission handler is different from the behaviour of the link click handler is that a click on a link is in essence an entirely different operation compared to a form submission. Upon clicking on a link, no extra data is submitted to the web server, and the result is always a navigation to a new web page, identified by the link. However, if a form is submitted, additional input data is sent to the server, which needs to be processed first, in order to determine if the current web page should be re-rendered, or if a new web page should be shown. An easy example is a login form: Here the server first needs to check if the submitted user credentials are valid in order to advance to the next page, or if they are invalid and the current page should be re-rendered including an appropriate error message. This explains why the form submission handler of *Mascherl* does not push a new URI to the browser history stack right away, because the handler has to wait for the server result first. If the server confirms the submitted input data and therefore determines a navigation to a new web page, the URI of this new web page is pushed to the browser history stack. However, if the submitted data is not valid, or if the current web page should be re-rendered because of some other reason, the browser history stack remains unchanged.

Employing the above behaviour for link clicks and form submissions makes sure that only *safe* interactions (i.e. navigations) are mapped to the browser history stack, whereas *unsafe* operations (i.e. form submissions) are not added to the browser history stack [139]. Therefore it is guaranteed that users do not unintentionally repeat unsafe operations just by using the browser

back and forward buttons.

Handling of browser back and forward buttons

In the event of a click on the browser back or forward buttons, *history.js* produces a *statechange* event, which is listened to by a JavaScript handler of *Mascherl*. This JavaScript handler triggers a partial AJAX request to the URI referenced in the event. The server side processing of this request and the following response handling part are then exactly the same as the corresponding parts in the link click scenario. Figure 3.12 illustrates the complete process.

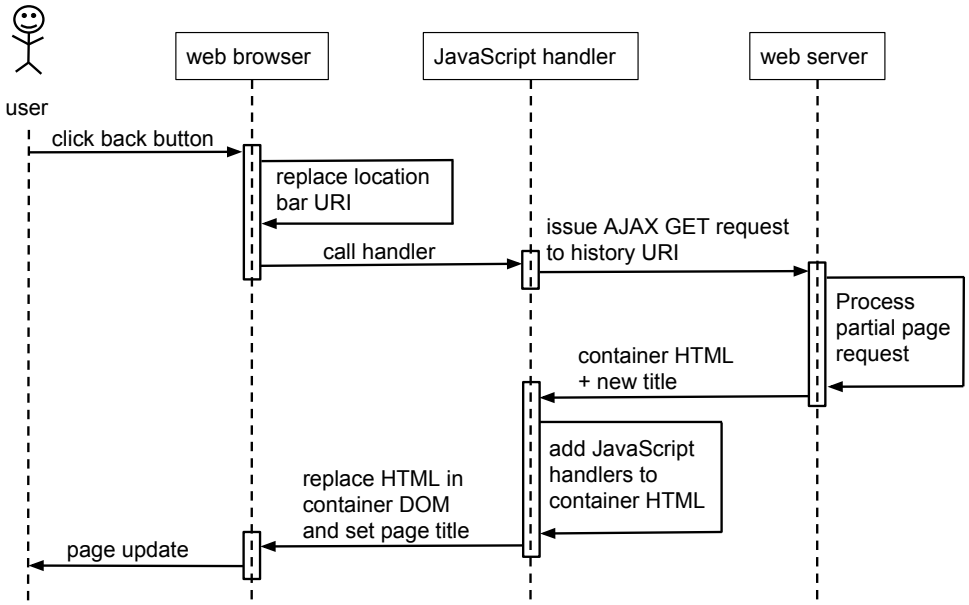


Figure 3.12: Sequence diagram of the process triggered by a click on the browser back button in a *Mascherl* based web application.

AJAX error handling

As communication errors can occur anytime, the JavaScript library of *Mascherl* clearly needs to implement some kind of AJAX error handling for the entire AJAX communication described above. In the default full page communication controlled by the browser, a browser specific error page is presented to the user in the case of a communication error. Since this site cannot be shown manually, a custom error page is shown instead. Actually, this error page is only an error fragment, which is pre-loaded on the full page request that loads the overall single page HTML interface of the web application. The contents of the error fragment can be specified on the main static HTML page using a custom <*script*> tag with an *id* attribute equal to *error*, and a *type* attribute equal to *text/x-error-template*, as already shown in Listing 3.1. If an AJAX

communication error occurs, this error fragment is set as the content of the main container in the DOM of the web page, and the page title is set to *Error*.

3.4 HTTP communication sub-protocol

The HTTP communication between client and server obeys a special, *Mascherl* specific sub-protocol, which aims two purposes:

- It enables the server to accurately distinguish between full page requests issued by the web browser, and partial page requests issued by the JavaScript library of *Mascherl*.
- It allows the client to correctly interpret the response received from the server, and thus to perform the appropriate actions.

Client-server communication sub-protocol

A full page request sent by the browser to the web server does not include any additional *Mascherl* specific HTTP headers or query parameters, but it is rather a plain HTTP request to the URI in the browser location bar. In contrast, every AJAX request issued by the JavaScript library of *Mascherl* adds such additional metadata.

Partial GET requests, which are the result of link clicks or clicks on the browser back or forwand buttons, are propagated with the following additional query parameters:

- *m-container* - the container which should be processed on the server, specified by the *m-container* attribute of the link that triggered the process.
- *m-page* - the current page group displayed in the browser.
- *m-app-version* - the version of the web application displayed in the browser.

The container parameter is obviously necessary for correct partial processing on the server, however, the other two parameters are also essential. The page group is needed on the server in order to determine if the container, which should be processed, is enough in order to display the full contents of the requested page, given the current page in the browser. If this is not the case, the web server automatically adjusts the container parameter to an appropriate parent container, possibly even to the outermost *main* container. Finally, the application version is needed in order to determine if the client uses an outdated version of the main static HTML page, and should therefore perform a full page request in order to reload the newest version of the web application. The idea for this versioning mechanism is taken from PJAX [52].

Partial POST requests, which are the result of form submissions, also add additional metadata to the HTTP request, however, instead of using query paramaters to transport the metadata, the following HTTP headers are added:

- *X-Mascherl-Container* - the same as *m-container*.
- *X-Mascherl-App-Version* - the same as *m-app-version*.

- *X-Mascherl-Form* - the *id* of the form that is submitted.

The page group is not needed on POST requests, because it actually only has an effect on requests triggered by the browser back and forward buttons.

The reason for using HTTP headers instead of additional query parameters here is that adding query parameters to POST requests, which already send parameters to the server in the HTTP body, can cause unwanted behaviour on the Java EE server, because the Servlet API does not differentiate between query parameters and HTTP POST body parameters, and processing both types of parameters in the same request is not implemented consistently among different servlet containers. Conversely, the reason for using query parameters on GET requests instead of additional HTTP headers is that then the potential full page requests issued by the browser have the same URI as the AJAX GET requests issued by *Mascherl*, which can confuse the browser cache, e.g. resulting in the display of a previously cached fragment version of the respective page (from an AJAX request) upon deep linking into that page.

Server-client communication sub-protocol

The server adds the following non-standard HTTP headers to partial response:

- *X-Mascherl-Title* - the new title of the web page.
- *X-Mascherl-Page* - the new page group.
- *X-Mascherl-Container* - the container, which has been processed, and whose markup is included in the reponse body.
- *X-Mascherl-Url* - the new URI (optional).
- *X-Powered-By* - always set to `Mascherl x.y.z`, with x.y.z referring to the used version of *Mascherl*.

These headers are used by the JavaScript library of *Mascherl* for correctly handling the response, as described in the previous section.

3.5 Model evaluation approach

Like many other web application frameworks, *Mascherl* promotes the model-view-controller (MVC) pattern. However, instead of using the common MVC pull based approach, *Mascherl* implements MVC push. The difference between these two approaches is illustrated in Figure 3.13.

In a MVC pull based architecture the model is used as a facade for accessing all kinds of backend systems, like e.g. databases or remote services, but also all kinds of contextual information from the server (e.g. HTTP request headers, HTTP session attributes, etc). The data, which is effectively used by the view is thereby pulled from the various systems using the model object during the actual rendering process.

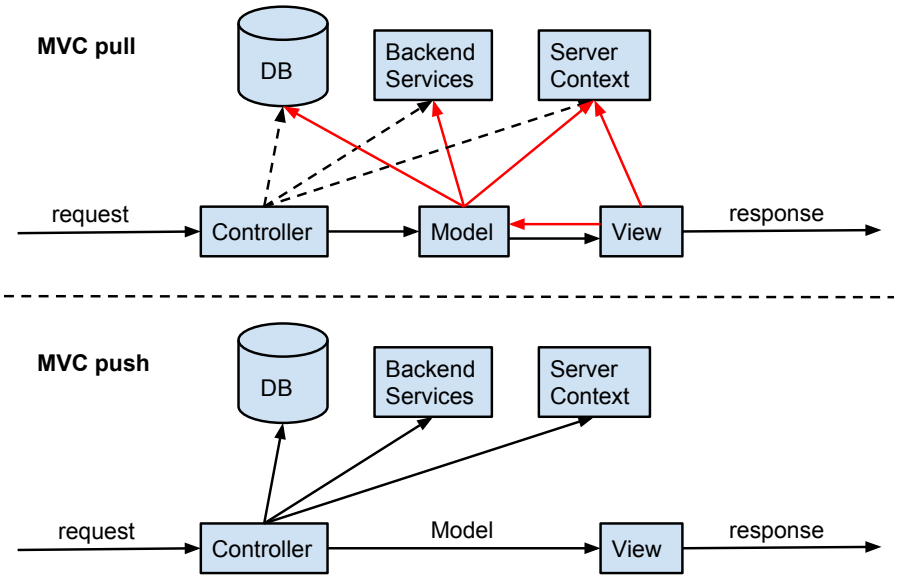


Figure 3.13: Comparison of MVC pull and push approaches, showing a much better separation of concerns in MVC push.

In an MVC push based architecture the sole purpose of the model is to transfer data from the controller to the view. The model object has no connections to any other systems or remote objects, and thus the entire data which is needed by the view must be added to the model by the controller, before passing it over to the view. The data is therefore pushed into the view.

A great number of web application frameworks promote MVC pull over MVC push, because it enables easier application development, with the drawback of entangling view, model, and controller code. From the frameworks presented in Section 2.3, JSF uses a pure MVC pull based approach using the Java Expression Language (EL) for pulling the data into the view, whereas Spring Web MVC, JSR 371 MVC, and the Play Framework use a combination of MVC push and pull. This combination is achieved by providing an explicit model object to the view (e.g. using *org.springframework.web.servlet.ModelAndView* in Spring Web MVC, or *javax.mvc.Models* in JSR 371 MVC) on the one side, and allowing to access all kinds of beans and server objects using the Java EL on the other side.

Parr [104] motivates the usage of MVC push over MVC pull, because it enforces a much better separation of concerns between the different components of the web application. García et al. [43] furthermore introduce a double-model approach, in order to implement MVC push. The term double-model refers to using separate model objects in the controller for accessing the backend systems, and then transforming those model objects into different model objects for the view using a transformation process.

Mascherl uses a pure MVC push architecture in order to completely detach controller logic from the view rendering process, and to prevent the view from executing any logic. The framework even goes one step further by dividing the overall model for the whole view into separate

per-container models, i.e. every container gets its own model. This separation enables isolated processing of every container inside the same view. Furthermore, this utilization of MVC push allows to predetermine the entire data needed for rending each separate container of a view, and this allows to prevent worst case scenarios, i.e. a chain reaction of transitive lazy model evaluations. Nevertheless, it must be recognized that this programming model enforces additional constraints on the web application developer, and thus makes the actual development more challenging.

3.6 Server-side partial-page rendering implementation

The Java based web frameworks presented in Section 2.3 primarily promote Java Server Pages (JSP) for rendering the HTML view, because JSP has been the main rendering system in Java EE for a long time. However, all of these frameworks also endorse other rendering technologies: JSF was initially built upon JSP, but as of JSF 2.0, JSP was replaced with Facelets [78]. Spring Web MVC also supports Velocity and Freemarker templates among others [124]. JSR 371 MVC is designed to support even more different rendering technologies and templating systems, including Velocity, Freemarker, Handlebars, Mustache, Facelets, and Thymeleaf [51]. Finally, the Play Framework provides its own Scala-based template engine, which is inspired by ASP.NET Razor [110].

Apart from that, the single-page JavaScript web frameworks presented in Section 2.2 also follow different rendering approaches, however, instead of producing the final HTML on the web server, the rendering is executed in the client browser in JavaScript. While AngularJS has its own built-in templating engine [5], EmberJS makes use of Handlebars [32]. The exception here is GWT, because it uses a component based approach, and thus does not directly use templates for producing the HTML.

Moreover, some popular websites published their approach for rendering the HTML view. Google uses *Closure* templates for Google Plus [65], Twitter employs *Mustache* [50], and LinkedIn utilizes *Dust.js* [89]. In general, a great number of novel rendering frameworks has emerged in the last years. A comparison of some important representatives has been performed by LinkedIn [89].

This abundance of novel rendering frameworks and template engines is a consequence of the shortcomings of traditional HTML rendering technologies, like JSP. Furthermore, new requirements have emerged, which can not be fulfilled by existing technologies. Above all, the need for running in the client browser, and therefore the requirement for the implementation to be in JavaScript is the most important one for client side web frameworks. Apart from that, novel template engines are very restrictive regarding the usage of logic in templates. In JSP it is possible to execute code snippets inside a view template, because JSP allows to include normal Java code in the view template. These powerful capabilities are often misused, leading to a tight entanglement of view, model and controller logic. Some of the novel rendering frameworks are therefore completely logic-less in order to enforce a strict separation between the view and the rest of the application. A prominent example of a logic-less templating engine is *Mustache* [100, 101], which is implemented in over 30 different programming languages, including Java and JavaScript.

Mascherl utilizes the Java implementation of *Mustache* for producing the HTML markup out of the current model objects and the associated *Mustache* based template files. As discussed before, a MVC push based mechanism is used to push the model from the controller into the view. The view, i.e. the *Mustache* based implementation of the view in *Mascherl*, then uses this model for evaluating the value references of the view template, without executing any logic from the view template. As a consequence, the view is separated from the rest of the application, with the model acting as a one-way interface between view and controller. Therefore, no logic can be put into the view or the model of a *Mascherl* based web application, and no lazy evaluation of model values can be performed.

However, *Mascherl* cannot just use the Java implementation of *Mustache* directly, because *Mustache* does not support partial-page rendering out of the box. Therefore, additional logic needs to be added in order to support the partial-page rendering approach of *Mascherl*, i.e. handling containers and fragments in view templates.

The prototype of *Mascherl* uses a file based approach for splitting up templates into multiple fragments. This means that every page fragment, which should have the abilitity of being rendered independently from all other parts of a view, must be placed into a separate file. In the case that a respective page fragment is rendered independently, this has the effect that the view rendering engine only has to process this one file, instead of having to parse a big template file and extracting the portion belonging to the respective page fragment. The idea for splitting up an entire page into separate files in order to be able to process these files independently is also presented in various papers [17, 21]. Moreover Figure 3.8, which is taken from [21], illustrates the approach.

The syntax for specifying a container in a *Mustache* based page fragment template in *Mascherl* is `{{{@mycontainer}}}`, where *mycontainer* is the name of the container. This container definition is already shown in Listing 3.1 for the definition of the *main* container in the global static HTML page. It can be compared to the *outlet* syntax used in the templating approach of EmberJS, e.g. `{{outlet "toolbar"}}` [33]. Furthermore, Listing 3.2 shows the page fragment template file *overview.html*, which defines the container *form*, and Listing 3.3 shows the associated template file *overviewform.html*, which is automatically included in the aforementioned container by *Mascherl*.

```
1 Overview page
2 <p>{{welcome}}</p>
3 <p><a href="/page1" m-container="main">Page 1</a></p>
4 <p><a href="/page2" m-container="main">Page 2</a></p>
5 {{{@form}}}
6 <p><a href="https://www.google.at" m-ignore>Link to Google</a></p>
7 Overview end
```

Listing 3.2: The *Mustache* based page fragment template file *overview.html*, which defines a container with the name *form* used for inclusion of sub-fragments.

```
1 {{message}}
2
3 <div>
4   <form id="overview-form" method="post">
5     <p><input type="text" placeholder="Vorname" name="firstname" /></p>
```

```
    <p><input type="text" placeholder="Nachname" name="lastname" /></p>
    <p><input type="submit" /></p>
  </form>
</div>
```

Listing 3.3: The *Mustache* based page fragment template file *overviewform.html*, which can be included into the *form* container specified in Listing 3.2.

If the above template files are rendered by *Mascherl*, the output shown in Listing 3.4 is produced, given an appropriate model containing the respective values for *welcome* and *message*.

```
Overview page
<p>Welcome text from the model.</p>
<p><a href="/page1" m-container="main">Page 1</a></p>
<p><a href="/page2" m-container="main">Page 2</a></p>
<div id="form" m-page="OverviewPage">
  The sample message from the model.

  <div>
    <form id="overview-form" method="post">
      <p><input type="text" placeholder="Vorname" name="firstname" /></p>
      <p><input type="text" placeholder="Nachname" name="lastname" /></p>
      <p><input type="submit" /></p>
    </form>
  </div>
</div>
<p><a href="https://www.google.at" m-ignore>Link to Google</a></p>
Overview end
```

Listing 3.4: The HTML output produced by the *Mustache* based view engine of *Mascherl*, if the template files of listings 3.2 and 3.3, and an appropriate model are supplied.

Listing 3.4 shows that the page fragment file *overviewform.html* is evaluated and set as the content of the *form* container. Additionally, the content is wrapped into a <*div*> element, which specifies the container name as *id* of the element, and the page group of the request in the *m-page* attribute. This wrapper along with its attributes is then used by the JavaScript library of *Mascherl* in the client browser for correctly replacing the content of the *form* container, if necessary. The JavaScript library also parses the links in the HTML, i.e. the <*a*> elements, which are not marked with *m-ignore* (like the external link to Google in the example), and adds the click handler presented in Section 3.3 to them. Finally, the form element is also parsed, and the corresponding submission handler is added to it. The example presented here deliberately does not include the specification of the relation between the two page fragment template files *overview.html* and *overviewform.html*, because this concept is explained in the following section.

Even though the prototype of *Mascherl* uses a view implementation based on the *Mustache* Java rendering engine, the web framework is designed to be independent of any view technology. Therefore, *Mascherl* specifies an internal API for its view engine, which is currently only implemented for *Mustache*, however, it is easy to change this implementation to another one, e.g. one for Handlebars, or Dust.js. The view engine implementation therefore needs to provide two methods, one for rendering a full page, and one for rendering only one container of a page,

given the associated model objects. How this is accomplished internally is left to the view implementation, although it should be noted that for rendering only a container of a page it is not desirable to render the entire page and then only extract the requested container's markup out of it, because this has negative impacts on the performance of the framework, and in addition, the model objects for the parts outside of the specific container are not available.

3.7 Integration with Java EE supporting partial model evaluation

For the implementation of the web framework prototype on top of Java EE there are basically two alternatives:

- Build the framework on top of the Servlet API by creating a servlet that takes care of low-level request and response handling, like e.g. done in JSF.
- Implement the framework using a higher level Java EE API, which allows for a more high-level request and response processing, but at the same time imposes additional restrictions on the implementation. An example for this alternative is JSR 371 MVC, which is built upon JAX-RS.

Initially, the prototype has been built upon the Servlet API, however, after some time it has become apparent that this approach results in a great number of duplicate features, which are already present in Java EE, like e.g. request path matching, parameter parsing, and request body parsing. In Java EE all these functionalities are already provided by JAX-RS, thus the prototype of *Mascherl* is now also based on JAX-RS. Concretely, *Mascherl* is built upon the JAX-RS 2.0 implementation from Apache, which is part of the *Apache CXF* project [7]. However, the prototype adheres to the official JAX-RS API as much as possible, in order to be able to migrate to a different JAX-RS vendor, e.g. *Jersey* from the GlassFish project [81]. Nevertheless, there are some features in the prototype that are based on the API of *Apache CXF* instead of the official JAX-RS API, because the JAX-RS API does not expose the desired functionality (yet).

Even though the JAX-RS API provides multiple extension points, the prototype of *Mascherl* uses a different integration approach: it uses an interface with default methods, which is a new feature of Java 8. Therefore, the controller classes of a *Mascherl* based web application have to implement the interface *MascherlPage*, which is shown in Listing 3.5.

```
1  public interface MascherlPage {
2
3      @POST
4      @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
5      @Produces(MediaType.TEXT_HTML)
6      public default Response post(@Context HttpServletRequest request,
7                                   @Context HttpServletResponse response,
8                                   @HeaderParam(X_MASCHERL_FORM) String form,
9                                   @HeaderParam(X_MASCHERL_CONTAINER) String
                                        container,
10                                  @HeaderParam(X_MASCHERL_PAGE) String page) {
11         // implementation cut for clarity
12     }
```

```
13      @GET
14      @Produces(MediaType.TEXT_HTML)
15      public default Response get(@Context HttpServletRequest request,
16                                  @QueryParam(M_CONTAINER) String container,
17                                  @QueryParam(M_PAGE) String page) {
18          // implementation cut for clarity
19      }
20
21      public default String getTitle() {
22          return getClass().getSimpleName();
23      }
24  }
```

Listing 3.5: The interface *MascherlPage*, which must be implemented by all controller classes in a web application based on the prototype of *Mascherl*.

The default methods *get* and *post* are recognized by JAX-RS using the JAX-RS specific annotations. Both methods implement the full request handling of *Mascherl* as outlined in Section 3.1 for HTTP GET and POST requests, respectively.

For GET requests, the implementation first determines if the request is a full page request or a partial page request, and then invokes the corresponding method on the view engine of *Mascherl*, presented in the previous section. In addition, a reference to the controller instance is passed to the view engine, in order to perform the necessary model evaluations. To find the associated method, which assigns a page fragment template file to a specific container as well as calculates the model values for that container, the annotation *@Container("containerName")* is used. An instance of this annotation specifying the correct container name needs to be added to the respective container method in the controller class. The return value of that method must be of type *Partial*, which contains the name of the page fragment template file and all evaluated model values for the associated container. Due to the fact that every container of the current page has an associated method in the page's controller class, the page fragment template file and the corresponding model values for each container can be calculated independently from all other containers of the same page. Thus, this approach enables container-based partial model evaluation.

The handling of POST requests is considerably different from that of GET requests, because POST requests correspond to actions that are performed using the submitted data. To find the action associated with a form submission, *Mascherl* uses the annotation *@FormSubmission("formId")*, which must be added to the related action method in the controller class, while specifying the correct form id. The action method can access the data submitted as part of the request by using standard JAX-RS annotations, in particular *@BeanParam* and *@FormParam*. Moreover, various return types are supported: *String*, *URI*, *Class*, and the *Mascherl* specific type *ContainerRef*. Any of the first three types results in an internal request dispatch to the associated URI, which is determined by calling the *UriBuilder* of JAX-RS. This dispatch performs an internal partial-page GET request, rendering the *main* container of the specified URI. In contrast, a result type of *ContainerRef* means that the referenced container of the current page should be re-rendered. In order to achieve that, no request dispatch is necessary, because the *post* method can directly call the *get* method of the current controller class.

Listing 3.6 shows a sample controller class from a web application based on the prototype of *Mascherl*. This controller class specifies two containers, *main* and *form*, as well as one action method for the form with an id equal to *overview-form*. The two container methods reference the page fragment template files *overview.html* and *overviewform.html* shown before in listings 3.2 and 3.3, respectively.

```
1  @Path("/")
2  public class OverviewPage implements MascherlPage {
3
4      private String message;
5
6      @Override
7      public String getTitle() {
8          return "Overview";
9      }
10
11     @Container("main")
12     public Partial main() {
13         return new Partial("/templates/overview.html")
14                 .set("welcome", "Welcome to Mascherl!");
15     }
16
17     @Container("form")
18     public Partial form() {
19         return new Partial("/templates/overviewform.html")
20                 .set("message", message);
21     }
22
23     @FormSubmission("overview-form")
24     public ContainerRef submit(@BeanParam OverviewForm overviewForm) {
25         message = "Hello " + overviewForm.getFirstname() +
26                 " " + overviewForm.getLastname();
27         return new ContainerRef("form");
28     }
29 }
```

Listing 3.6: The controller class *OverviewPage* of a web application based on the prototype of *Mascherl*. The class shows the definition and implementation of two container methods and one form action method.

As shown in the above listing, the controller class *OverviewPage* is mapped to the root path of the web application using the *@Path* annotation from JAX-RS. Furthermore, the class specifies the page title for the associated web page, i.e. *Overview*. Finally, the form action method *submit()* uses the class *OverviewForm* as a parameter in order to encapsulate the values submitted in the associated POST request. However, the actual mapping of form input fields to variables is effectively performed by JAX-RS using the *@FormParam* annotation. Listing 3.7 shows the class *OverviewForm*.

```
1  public class OverviewForm {
2
3      @FormParam("firstname")
4      private String firstname;
```

```
5       @FormParam("lastname")
6       private String lastname;
7
8       // getters and setters
9   }
```

Listing 3.7: The class *OverviewForm* used in the form action method of the controller class shown in Listing 3.6 for mapping and accessing the values submitted in the current POST request.

In order for the above annotation based approach to work, the web framework needs to have an annotation index of all classes of the web application. This index is built upon startup of the web application by using an implementation of *ServletContainerInitializer*, which finds all classes in the classpath of the current web application that are annotated with the JAX-RS *@Path* annotation. After checking if the referenced classes really do implement the interface *MascherlPage*, the annotations of the respective classes are scanned and added to an index in the class *MascherlContext*. This singleton context class is then used by the *get* and *post* methods of *MascherlPage*, and by the implementation of the view engine, in order to properly process the HTTP requests issued to the framework, as presented above.

Finally, Listing 3.8 shows the interface *MascherlRenderer*, which defines the internal view engine API of the prototype of *Mascherl* that is used by the methods of *MascherlPage* presented above. As discussed in the previous section, *Mascherl* includes an implementation of this API that is based on the Java implementation of the *Mustache* templating engine. Both methods of the API take a *MascherlPage* instance, and return a JAX-RS *Response* object, which is used by the underlying JAX-RS framework to send the actual response to the client.

```
1   public interface MascherlRenderer {
2
3       public String FULL_PAGE_RESOURCE = "/index.html";
4
5       public Response renderFull(MascherlPage page);
6
7       public Response renderContainer(
8           MascherlPage page, String container, String clientUrl);
9
10  }
```

Listing 3.8: The interface *MascherlRenderer*, which specifies the internal view engine API of the prototype of *Mascherl*.

3.8 Adoption of Play's state management approach

As discussed in Chapter 1, the web framework has to avoid using traditional server-side state saving mechanisms, like e.g. HTTP sessions, in order to constitute a stateless web server that can be scaled horizontally without further considerations. To achieve this, the state management approach found in the Play Framework should be utilized.

The idea of Play's state management approach is to use an encrypted HTTP cookie to store the state on the client, as already discussed in Section 1.1. The actual implementation of this approach can be found in the Scala sources of the Play Framework on GitHub, especially in the files *Crypto.scala* [53] and *Http.scala* [54].

As can be seen in the above source files, Play uses *AES* for encryption and decryption of the session cookie values. The exact AES method can be specified using the *application.crypto.aes.transformation* config parameter. Previously, Play used *AES/CBC/PKCS5Padding*, however, as of Play 2.4 *AES/CTR/NoPadding* is used by default. The secret key, which is needed by AES, is determined from the application secret, which is a string that can be configured using the *play.crypto.secret* config parameter. From this string a *SHA-256* message digest is calculated, and the result is used as the key for AES. Obviously, all deployed instances of the same Play based web application have to share the same application secret in order to ensure that all instances use the same key for AES. Otherwise, different instances of the web application can not decrypt each others' session data, thus breaking the horizontal scalability of the application.

Mascherl uses the exact same approach as the Play Framework for encryption and decryption, and for the generation of the key for AES. The application secret in *Mascherl* can be configured using the config parameter *org.mascherl.session.secret*, which is set to the following default value: "This is mascherl's secret session key for development. Changing this for production is a MUST." The default value is self descriptive, and correctly states that the application secret must be changed for a production deployment of the application, otherwise any attacker that gains access to the session cookie of any user is able to read this cookie, and even worse, any attacker could fake session data of the application. Therefore it is essential that the application secret really is kept secret.

The session implementation of *Mascherl*, however, deviates from the session implementation of the Play Framework regarding the types of allowed session data. Play only allows string values to be stored in the session, whereas *Mascherl* allows all types that can be serialized into JSON by the *Jackson* library [48], i.e. all primitive Java types, strings, various date types, and any kind of plain old Java objects (POJOs).

For encryption, all session data fields along with their string based keys are passed to *Jackson*, which creates a serialized JSON string containing all data. This string is then encrypted using AES and the secret key calculated from the configured application secret. Finally, the outcome is base-64 encoded and added to the HTTP response as a HTTP cookie using the name "mascherl". On the subsequent HTTP request, the browser automatically adds this cookie to the request header. The web application server can then base-64 decode the cookie value, decrypt it using AES with the same key as used before, and finally deserialize it using *Jackson*. Afterwards, the application can access all data stored in the session on previous requests using the correct type. In this process, no type information is added to the serialized JSON. The application is expected to provide the associated type of a value upon deserialisation. For that, the class *MascherlSession* provides the necessary access methods, as shown in Listing 3.9.

```
1 public class MascherlSession {
2
3     public static MascherlSession getInstance() {}
4
5     // constructors cut for clarity
```

```
6       public void remove(String key) {}
7
8       public void put(String key, Object data) {}
9
10      public Integer getInt(String key) {}
11
12      public String getString(String key) {}
13
14      public Date getDate(String key) {}
15
16      public <T> T get(String key, Class<T> expectedType) {}
17
18      public boolean contains(String key) {}
19
20      public String serialize() {}
21
22      // the rest is cut for clarity
23  }
```

Listing 3.9: The class *MascherlSession*, which provides various methods for adding, accessing, and removing session data. The implementation of all methods is cut for clarity.

The current instance of *MascherlSession* can be accessed using the static method *getInstance()* within the current thread of the servlet container processing the HTTP request.

3.9 Conclusion

The prototype of *Mascherl* presented in this chapter enables RESTful, stateless, single-page web application development in Java using partial model evaluation and partial page rendering. RESTful URIs are supported by building upon the established JAX-RS standard of Java EE, which is the standardized way of creating RESTful web services in Java. The framework further promotes stateless web servers by providing a custom session implementation that allows to store client related state data in the browser rather than on the web server, which enables horizontal scalability without web server synchronization. The prototype tightly integrates the HTML5 history API into its Hijax-based mechanism for request and response handling. This mechanism effectively transforms an otherwise multi-page interface into a single-page web application, without revealing it to the user. The user therefore does not recognize any differences to a traditional multi-page application, apart from the application responding faster, resulting from reduced server load and decreased network traffic. This speedup is possible because of the usage of partial requests, which only cause partial model evaluations and page fragment rendering on the server, instead of triggering full page processing for every web request.

All common methods of interaction with web pages are supported by the web framework. They include link navigations, form submissions, the browser back- and forward buttons, and even deep linking to any page of the web application. The general idea on how to support all these interactions using partial page requests, as well as their underlying processing sequences, and the approach for AJAX error handling have been presented in this chapter. Moreover, the

HTTP sub-protocol used to identify partial page requests and to transport additional metadata has been discussed.

This chapter also proposes a method for implementing true partial model evaluation and partial page rendering without redundant processing of controller logic or page templates that do not contribute to the current HTTP response. An approach for the elemental structuring and organization of page fragment templates that enables partial page rendering as well as a technique for separating controller logic and model calculation on a per-container basis have also been presented.

Finally, a strict separation of model, view, and controller logic is ascertained by motivating the usage of MVC push over MVC pull, and by using a logic-less template language that avoids the introduction of any logic into view templates, which essentially belongs into the controller. The entanglement of model, view, and controller code that is often found in JSP based web applications is therefore prevented.

Nevertheless, the prototype of the web framework *Mascherl* still shows some weaknesses, which are addressed in the following chapter.

CHAPTER 4

Refinement of the prototype

In this chapter, the prototype of *Mascherl* presented in the previous chapter is evaluated, and then the work on the prototype is elaborated in order to create a production ready version of the web application framework *Mascherl*.

4.1 Evaluation of the prototype

Even though the prototype of *Mascherl* can already be used to build RESTful, stateless, single-page web applications that employ partial model evaluation and page fragment rendering, it still has major shortcomings, which lead to cumbersome web application development.

Certainly, the fact that every web page needs to have a separate controller class is annoying to web application developers, because web pages that share various functionalities must be separated into two classes, instead of using one class that provides the controller logic for both web pages. To circumvent this problem, class hierarchies can be introduced, however, leading to further problems as soon as multi-inheritance becomes necessary.

Another limitation of the prototype of *Mascherl* is the lack of support for typesafe navigation, like e.g. available in the Play Framework. In the prototype, links and forms reference their associated targets using the respective URI of the target, as e.g. shown in Listing 3.2. However, if some target URIs are refactored, it is not guaranteed that all links and forms are updated too, because the references are not typesafe. Therefore, *Mascherl* should provide support for typesafe target references from view templates to controllers.

Moreover, the fragmentation of the page templates into many separate files, one for every page fragment that can be put into a container, is undesirable, because above all a web designer does not deliver page templates in fragmented files, and thus the web application developer has to conduct this separation. Furthermore, if a web page contains a great number of parts that need to be updated independently of each other, the number of template files gets very big, and therefore the developer can easily lose track of the myriad of page fragment template files. Additionally, the view template language *Mustache* used in the prototype actually provides syntax for defining

different page sections inside one template file, which can be used to implement page fragments for *Mascherl* for a whole web page in only one template file, rather than using various separate files.

The fact that the prototype of *Mascherl* does not make any use of the extension points provided by JAX-RS is definitely not severe, however, utilizing these extension points avoids the compulsory interface for controller classes, i.e. *MascherlPage*, which is preferable.

The prototype further does not integrate any additional Java EE technologies, like e.g. Bean Validation, which is proven to be very helpful in applications that heavily rely on data input and output.

Any JavaScript used by the prototype of *Mascherl* neither is organized in modules using AMD (asynchronous module definition), nor uses any kind of optimization, minification, or obfuscation methods, and therefore is not recommended for productive web environments. In addition, the JavaScript library of *Mascherl* does not provide any extension points.

Finally, there is still no support for declarative asynchronous request handling, which is required for building a real non-blocking web server, as proposed in Chapter 1.

All of the above limitations are addressed in the refinement of the prototype of *Mascherl* discussed in the following.

4.2 An advanced approach for page fragment templating

As mentioned in the previous section, the *Mustache* template engine supports page section based templating in combination with template inheritance, and this functionality can be used to implement page fragments for *Mascherl*. However, it has to be noted that this functionality is currently not included in the official *Mustache* specification, although it is under heavy discussion, and major implementations have already added support for it, including the Java implementation used in *Mascherl* [49]. Listing 4.1 shows the proposal for template inheritance in *Mustache*, as it is implementated in the Java version of the *Mustache* template engine.

```
1  super.mustache:
2      <html>
3      <head><title>{{$title}}Default title{{/title}}</title></head>
4      <body>
5      {{>navigation}}
6      <div class="content">
7      {{$content}}Default content of the page{{/content}}
8      </div>
9      {{>footer}}
10     </body>
11     </html>
12
13 sub.mustache:
14     {{<super}}
15     {{$title}}Profile of {{username}} | Twitter{{/title}}
16     {{$content}}
17     Here is {{username}}'s profile page
18     {{/content}}
19     {{/super}}
```

```
20 the virtual mustache template that is rendered:
21     <html>
22     <head><title>Profile of {{username}} | Twitter</title></head>
23     <body>
24     {{>navigation}}
25     <div class="content">
26     Here is {{username}}'s profile page</div>
27     {{>footer}}
28     </body>
29     </html>
```

Listing 4.1: Proposal for template inheritance in *Mustache* taken from GitHub Gist [46]. The listing contains three files. The first one, *super.mustache*, is the super template of the example, the second one, *sub.mustache*, is the template extending *super.mustache*, and the third file is the effective virtual template that is rendered.

The above listing shows that the super template defines the basic structure of the page as well as replaceable blocks (or sections), that can be overwritten by sub templates. The syntax used for defining those sections is `{{$sectionName}}content{{/sectionName}}`. The super template can define multiple such sections, and also set their default content. Any sub template extending a super template can overwrite the content of every section defined in the super template, but it does not have to. However, no sub template can contribute content outside of a section to the effective template.

To improve the templating approach of *Mascherl*, those sections shall be used as the page fragments that make up a *Mascherl* web page, instead of dividing a page template into different files and using the *Mascherl* specific `{{{@containerName}}}` syntax introduced in the previous chapter.

In order to implement this new approach, the Java implementation of *Mustache* needs to be integrated much tighter into *Mascherl*, because *Mascherl* needs to access the different sections of every page template without processing the rest of the template to guarantee true partial page processing. Fortunately, the basic Java implementation of *Mustache* separates the different sections of a page template automatically, and additionally, also caches them separately. Thus, once an effective page template has been built up for the first time, all separate sections of this template can be accessed and rendered independently of each other, without having to parse the initial template file ever again, which is exactly what is required by *Mascherl*.

Apart from this new definition of page fragments, additional functionality must be added to the *Mustache* template engine in order to process the static HTML template of the web application properly. This template now also defines the main container using the new syntax, i.e. `{{$main}}{{/main}}`.

Listing 4.2 shows the page fragments, which are already shown in listings 3.2 and 3.3, however, using the new approach for page fragment templating. The abilities from *Mascherl's* point of view are the same here as in the previous chapter, because either the whole template or only the *form* container defined in the template can be processed and rendered independently. This is a striking fact, because the content of the *form* container is not moved into a separate file, and still *Mascherl* is able to process the page fragment separately, without touching the parts outside of the *form* container.

```
1  Overview page
2  <p>{{welcome}}</p>
3  <p><a href="/page1" m-container="main">Page 1</a></p>
4  <p><a href="/page2" m-container="main">Page 2</a></p>
5  {{$form}}
6    {{message}}
7
8    <div>
9      <form id="overview-form" method="post">
10       <p><input type="text" placeholder="Vorname" name="firstname" /></p>
11       <p><input type="text" placeholder="Nachname" name="lastname" /></p>
12       <p><input type="submit" /></p>
13     </form>
14   </div>
15 {{/form}}
16 <p><a href="https://www.google.at" m-ignore>Link to Google</a></p>
17 Overview end
```

Listing 4.2: The combined page template of listings 3.2 and 3.3 from the previous chapter, which uses the new syntax for page container and fragment definition in *Mascherl*. The contents of the two separated files from the previous chapter are now merged into one template file.

As a consequence of the new approach for page fragment templating, the controller classes of *Mascherl* have to be changed, because now only one page template is used for a whole web page, instead of defining one page fragment template for every container of the web page. How this is accomplished is explained in the following section.

4.3 Elaboration of the JAX-RS integration

The evaluation of the prototype of *Mascherl* pointed out that enforcing a separate class for every web page controller is a major deficiency. Therefore, a different approach is needed, which on the one hand allows to partition the controller logic on a per-container basis for every web page, but on the other hand does not dictate separate controller classes for every web page.

The solution for this problem in *Mascherl* is to use one method per web page that returns an instance of the class *MascherlPage*, which consists of the following parts:

- The page template file for the associated page.
- The title for the associated page.
- A callback function for every container of the page, which is able to calculate the model of that container.

Every web page is now referenced by one method in *any* controller class of the web application, and every container of that page is now associated with a callback function that calculates the model of that container. Thus, it is now possible to merge the logic for multiple web pages in one (controller) class. Additionally, the models of the various containers of every web page can still be calculated separately by using the respective callback functions. Therefore, true partial

model evaluations are still possible with the new approach, which is an important requirement for *Mascherl*.

Moreover, because of the fact that *Mascherl* uses Java 8, the aforementioned callback functions can actually be implemented using lambdas, instead of using anonymous inner classes or other constructs, which greatly improves readability of the controller source code.

However, the handling of form submissions has not yet been discussed. Previously, they were implemented as an annotated method in the controller class of the associated web page. In the new approach, form submissions are still handled by separate methods in the controller class, although they are now required to return an instance of the class *MascherlAction*. This class contains instructions for either navigating to a new page, or staying on the current page and re-rendering a certain container. In addition, the page definition of the target page, i.e. the instance of *MascherlPage*, can be supplied. If the page definition is omitted, an internal redirect to the target page is issued automatically.

As part of the changes to the controller definition of *Mascherl*, a new approach for the JAX-RS integration with the controller classes is used. In the prototype, every controller class is required to implement a certain interface, which provides the actual methods for handling HTTP requests. In the new approach this interface is not necessary anymore, however, every method of a controller class that corresponds to a web page definition must now be annotated with *@GET*, and every method that corresponds to a form submission must now be annotated with *@POST*. Additionally, all of these methods must be annotated with *@Path*, specifying the HTTP request path that maps to the respective web page or form submission.

Furthermore, the rendering is now triggered by a special implementation of the JAX-RS extension point interface *MessageBodyWriter*, and JAX-RS specific request and response filters are used to carry out various tasks, which are required before and after the execution of controller code, respectively. As a consequence, *Mascherl* is now much tighter integrated with JAX-RS.

Listing 4.3 shows the new version of the controller class *OverviewPage*, which has already been presented in the previous chapter in Listing 3.6.

```
1  public class OverviewPage {
2
3    @GET
4    @Path("/")
5    public MascherlPage overview() {
6        return Mascherl.page("/templates/overview.html")
7            .pageTitle("Overview")
8            .container("main", (model) ->
9                model.put("welcome", "Welcome to Mascherl!"))
10           .container("form");
11   }
12
13   @POST
14   @Path("/")
15   public MascherlAction submit(@BeanParam OverviewForm overviewForm) {
16     String message = "Hello " + overviewForm.getFirstname() +
17         " " + overviewForm.getLastname();
18     return Mascherl.stay().renderContainer("form").withPageDef(
19         overview()
20             .container("form", (model) -> model.put("message", message)));
```

```
21    }
22  }
```

Listing 4.3: The new version of the controller class *OverviewPage*, which shows the new container integration approach of *Mascherl*. The underlying page template is shown in Listing 4.2 above. For comparison, the old version of this class is shown in Listing 3.6.

The new version of the class *OverviewPage* is functionally equivalent to the old version of the same class, which is used in the prototype. It uses the page template shown in Listing 4.2, sets the page title to *Overview*, and defines two containers *main* and *form* with their respective model calculation lambdas. The form submission is handled in the method *submit()*, where a message is created, and finally, the instructions for re-rendering the container *form* of the current page with an overridden model containing the calculated message are returned.

The above listing further shows that every method is now annotated with the respective JAX-RS annotations for the associated HTTP method, i.e. *GET* or *POST*, as well as a *@Path* annotation. This means that the whole class is not bound to a specific request path as it is the case in the prototype, and therefore additional methods specifying different request paths for other web pages can be added to the same class.

Another consequence of the changes in the controller mapping is that HTML forms are now not automatically submitted to the HTTP path of the associated web page while using the id of the form element to find the associated action method. The request path of the form submission can rather be specified independently using the HTML form's *action* attribute. On the server, the associated action method is matched to the request only by using the given request path. The id of the form element is therefore irrelevant for the server communication in the new version of *Mascherl*. Listing 4.4 shows an additional action method, which is added to the class *OverviewPage* shown above. The action method triggers a navigation to the path */page1* using an internal redirect, as opposed to the first action method of the same class shown in Listing 4.3, which triggers a re-rendering of a container of the current page.

```
1  @POST
2  @Path("/submitOther")
3  public MascherlAction submitOther(@BeanParam AnotherForm anotherForm) {
4    // process data of anotherForm
5    return Mascherl.navigate("/page1").redirect();
6  }
```

Listing 4.4: An additional action method, which is added to the class *OverviewPage* shown in Listing 4.3. This action method is mapped to the request path */submitOther*, which is different from the path of the web page displaying the form. Additionally, the action method triggers a redirect to a new page, instead of re-rendering parts of the current page.

Finally, the *MascherlRenderer* interface presented in Section 3.7 has been adapted, in order to conform with the new requirements imposed by the new approaches for controller classes and template rendering, and by the elaborated JAX-RS integration. Listing 4.5 shows the new version of the interface *MascherlRenderer*.

```
1  public interface MascherlRenderer {
2
3    public String FULL_PAGE_RESOURCE = "/index.html";
4
5    public void renderFull(
6        MascherlApplication mascherlApplication,
7        MascherlPage page,
8        ResourceInfo resourceInfo,
9        OutputStream outputStream,
10       MultivaluedMap<String, Object> httpHeaders) throws IOException;
11
12   public void renderContainer(
13       MascherlApplication mascherlApplication,
14       MascherlPage page,
15       ResourceInfo resourceInfo,
16       String actionPageGroup,
17       OutputStream outputStream,
18       MultivaluedMap<String, Object> httpHeaders,
19       String container,
20       String clientUrl) throws IOException;
21
22   public ContainerMeta getContainerMeta(
23       String pageTemplate,
24       String container);
25
26 }
```

Listing 4.5: The new version of the interface *MascherlRenderer*, used for integrating different view engines into *Mascherl*. By default, a *Mustache* based implementation of this interface is provided.

4.4 Typesafe navigation support

The lack of typesafe navigation support in the prototype of *Mascherl* has been pointed out in the evaluation. In order to implement this feature in the *Mustache* based default template engine of *Mascherl*, the post-substitution feature of *Mustache* templates can be used, which allows to process the content of a specific template section in a Java function before rendering.

Mascherl therefore provides the function `{{#url}}...{{/url}}` in the template context, which takes the fully qualified class name of the controller class plus the associated method name of the controller method, for which the URI path should be returned. This feature works for normal page definition methods as well as action methods, and can therefore be used in the *href* attribute of HTML links, and in the *action* attribute of HTML forms in the templates of the web application. Listing 4.6 shows an example of a HTML link and form, which references the methods *overview()* and *submit()* of the class *OverviewPage* in a typesafe manner.

```
1 <a href="{{#url}}org.mascherl.example.OverviewPage.overview{{/url}}"
2    m-container="main">Overview</a>
3
```

```
4 <form id="overview-form"
5       action="{{#url}}org.mascherl.example.OverviewPage.submit{{/url}}"
6       method="post">
7 </form>
```

Listing 4.6: A HTML link and a HTML form, which reference the methods *overview()* and *submit()* of the class *OverviewPage*, respectively, in a typesafe manner by using the template function *url*.

4.5 Integration of Bean Validation

Bean Validation is a specification in the Java Community Process (JSR-303 and JSR-349) [76], which defines a declarative validation system for Java based applications. The model objects of the application, i.e. the *beans*, can therefore be annotated with special Bean Validation annotations, which determine the allowed values of the respective bean properties. Upon input (or output) of such a bean object, automatic validation can be triggered, which may produce certain validation errors. These validation errors can then be further processed by the system, e.g. by showing appropriate error messages to the user. All of the Java based web frameworks presented in Section 2.3 have built-in support for Bean Validation, because it has proven to be very helpful in various applications. That is why *Mascherl* should also include support for it.

Unfortunately, the JAX-RS standard currently does not include support for Bean Validation, however, Apache CXF, which is the JAX-RS implementation used by *Mascherl*, does provide custom support for it [8]. *Mascherl* makes use of this CXF specific Bean Validation feature in order to implement its own support for Bean Validation. This means, however, that this functionality only works as long as Apache CXF really is used as the underlying JAX-RS implementation.

The integration of Bean Validation into *Mascherl* is influenced by JSR 371 MVC, Spring Web MVC, and the Play Framework, rather than by JSF, because JSF automatically aborts processing of the current request upon validation errors, and shows them to the user. All other frameworks, however, do not automatically terminate request processing, but rather supply the validation errors to the controller, so that the controller logic can react on the validation errors itself. Every framework therefore uses a special class containing those validation errors. Table 4.1 gives an overview of those classes.

Web framework	Class containing validation errors of Bean Validation
JSR 371 MVC	`javax.mvc.validation.ValidationResult`
Spring Web MVC	`org.springframework.validation.BindingResult`
Play Framework	`play.data.Form`
Mascherl	`org.mascherl.validation.ValidationResult`

Table 4.1: Overview of web frameworks an their respective classes, which are used to supply the validation errors of Bean Validation to the controller logic.

Mascherl uses the class *ValidationResult*, which provides a method for accessing the validation errors of Bean Validation of the current request. Additionally, the method *isValid()* can be used in order to check if the object contains any validation errors at all. The instance of *ValidationResult* for the current request is created automatically using a CXF specific in-interceptor, before the controller logic is invoked. The controller can then access this instance using `ValidationResult.getInstance()`. Moreover, Spring (or any other IoC container) can be used in order to inject the instance into the controller class. Listing 4.7 shows an example, in which an action method uses Bean Validation to validate the input submitted by the user.

```
1  @POST
2  @Path("/login")
3  public MascherlAction loginAction(@Valid @BeanParam LoginBean loginBean) {
4    ValidationResult validationResult = ValidationResult.getInstance();
5    User user;
6    if (validationResult.isValid()) {
7      user = loginService.login(loginBean.getEmail(), loginBean.getPassword());
8    } else {
9      user = null;
10   }
11   // use user ...
12 }
```

Listing 4.7: Action method in a controller class of *Mascherl*, which uses *ValidationResult* in order to check for validation errors of the input submitted by the user.

However, the support for Bean Validation is not activated by default in *Mascherl*, because the aforementioned in-interceptor has to be registered with CXF first. The necessary configuration for CXF is shown in Listing 4.8.

```
1 <jaxrs:inInterceptors>
2   <bean class="org.mascherl.validation.cxf.CxfBeanValidationInInterceptor" />
3 </jaxrs:inInterceptors>
```

Listing 4.8: Apache CXF configuration needed to activate the Bean Validation feature of *Mascherl* in CXF.

4.6 Declarative, asynchronous request handling

As motivated in Chapter 1, the web framework *Mascherl* should provide support for declarative, asynchronous request processing to handle a great number of concurrent client requests on a single machine. The support should further be declarative, as opposed to the manual support for asynchronous request processing, like e.g. provided by the Servlet API, in order to ease development of asynchronous controller logic.

The support for asynchronous request handling presented in this section is heavily influenced by the ones provided by Spring Web MVC and the Play Framework, as discussed in Section 2.3. Thus, *Mascherl* uses special return values in its controller methods in order to indicate that the current request should be handled asynchronously. These special return values contain a promise

that the actual return value, i.e. the *MascherlPage* or the *MascherlAction*, will be available at some later point in time, but it is unknown when. Furthermore, they provide the possibility to register a callback, which is called as soon as the result is available. As discussed before, Play uses the class *Promise*, and Spring Web MVC uses various classes, including the class *java.util.concurrent.CompletableFuture*, which is included in Java 8, in order to achieve the aforementioned.

Mascherl supports using the Java 8 standard class *CompletableFuture*, and additionally, allows to use *rx.Observable*, which is a class provided by the *RxJava* library [116]. This library has been created by Netflix by porting the original Reactive Extensions library of Microsoft (Rx) from .NET to Java [97, 130].

Reactive Extensions (ReactiveX) are libraries for various programming languages, including RxJava for Java, which utilize the *Observable* and the *Iterator* pattern, as well as some ideas from *functional programming*, in order to ease asynchronous program composition [93, 112]. Essentially, ReactiveX covers the same use cases as the *CompletableFuture* API of Java 8, however, providing a much wider range of features for composing different asynchronous streams, handling errors and timeouts, and more. In a blog post [130], Netflix announced that they exclusively use ReactiveX Observables for their service layer, in order to decouple service consumer and producer, and additionally, to grant the service producer more flexibility for the actual service implementation.

In order to implement asynchronous processing on top of JAX-RS, *Mascherl*, again, makes use of an Apache CXF specific API. Even though JAX-RS provides support for asynchronous processing itself by using the *@Suspended* annotation in combination with the class *AsyncResponse*, it does not allow to trigger asynchronous handling without using that annotation on the associated resource method. Therefore, *Mascherl* uses a custom implementation of *org.apache.cxf.jaxrs.JAXRSInvoker* for triggering asynchronous request processing using the internal Apache CXF API. This invoker checks if the returned value of the resource method is an instance of *CompletableFuture* or *rx.Observable*, and in that case starts a new asynchronous process in CXF for the current request. Therefore, a callback is registered on the given instance of *CompletableFuture* or *rx.Observable*, which resumes the response on a different container thread, after the actual result is available, or after an exception has occurred. Listing 4.9 shows the method *handleObservable()* of the class *CxfObservableInvoker* in *Mascherl*, which implements the asynchronous request processing in CXF for the case that a controller method returns an instance of *rx.Observable*.

```
1  private Object handleObservable(Exchange exchange,
2                                  Observable<?> observable) {
3    if (observable instanceof ScalarSynchronousObservable) {
4      ScalarSynchronousObservable syncObservable
5          = (ScalarSynchronousObservable) observable;
6      // as if the method returned the value directly
7      return new MessageContentsList(syncObservable.get());
8    } else {
9      // start asynchronous processing
10     AsyncResponseImpl asyncResponse
11         = new AsyncResponseImpl(exchange.getInMessage());
12     asyncResponse.suspendContinuationIfNeeded();
```

```
13      observable
14          .subscribe(
15              (entity) -> asyncResponse.resume(entity),
16              (error) -> {
17                  logger.log(Level.WARNING,
18                      "Resuming suspended request with exception", error);
19                  asyncResponse.resume(error);
20              }
21          );
22      // as if the method returned void
23      return new MessageContentsList(Collections.singletonList(null));
24    }
25  }
```

Listing 4.9: The method *handleObservable()* of the class *CxfObservableInvoker* in *Mascherl*. At first, the method checks if the *Observable* is scalar, i.e. the actual result can be read without blocking, and in that case directly returns the result. If, however, the actual result is not available yet, an asynchronous process is triggered within CXF by using an instance of the CXF-internal class *AsyncResponseImpl*. Therefore, the instance of *AsyncResponseImpl* is connected with the instance of *rx.Observable* by subscribing to it. The asynchronous response is then resumed in a different thread with the outcome of the calculation, or with an exception that occurred during the calucation process.

As mentioned above, any controller method of *Mascherl* can make use of asynchronous request processing simply by returning an instance of *CompletableFuture* or *rx.Observable*, which evaluates to the actual return value of that method, i.e. *MascherlPage* or *MascherlAction*. Listing 4.10 shows an example controller method, which returns an *rx.Observable* of *MascherlPage*, instead of directly returning the value.

```
1   @GET
2   @Path("/async")
3   public Observable<MascherlPage> asyncRequest() {
4     return asyncService.calculateResultAsync()
5       .map((result) ->
6           Mascherl.page("/pageTemplate.html")
7               .pageTitle("Asynchronous Response")
8               .container("main", (model) -> model.put("result", result)))
9       .timeout(2, TimeUnit.SECONDS);
10  }
```

Listing 4.10: An example controller method in *Mascherl*, which returns an instance of *rx.Observable* of *MascherlPage* instead of directly returning the *MascherlPage*. It therefore triggers asynchronous request processing, as shown in Listing 4.9. To construct the *rx.Observable*, the controller method uses an asynchronous service, which itself returns an *rx.Observable* of its result type. This result is then mapped to a *MascherlPage* by using the *map()* method with an appropriate lambda function. Additionally, a timeout of two seconds is specified for the service call, which triggers a *TimeoutException* that resumes the HTTP request, if the service does not respond in time.

The timeout applied in the above listing illustrates one of the additional features that are provided by Reactive Extensions. This timeout property is especially helpful, because it allows to define upper limits on the application's response times in a declarative way, which is otherwise very hard to accomplish in a highly concurrent environment.

Finally, due to the fact that the asynchronous request processing of *Mascherl* is built upon a CXF-internal API, it is not activated by default. Just like the Bean Validation feature presented in the previous section, this feature also has to be registered with CXF first, by using the configuration shown in Listing 4.11.

```
1 <jaxrs:invoker>
2   <bean class="org.mascherl.async.cxf.CxfObservableInvoker" />
3 </jaxrs:invoker>
```

Listing 4.11: Apache CXF configuration needed to activate the asynchronous request processing feature of *Mascherl* in CXF.

4.7 JavaScript modularization and optimization

In the evaluation of the prototype of *Mascherl* it has been pointed out that the JavaScript library, which is an integral part of *Mascherl*, because it is responsible for handling the partial page updates and the history management in the client browser, should be modularized and optimized in order to be made production ready. Therefore, *Require.js* [114] is used in order to structure the JavaScript code in modules using AMD (asynchronous module definition) [12], to apply minification and obfuscation methods, and to be finally able to serve the JavaScript among its dependent libraries asynchronously to the browser. All of these modifications help in speeding up the initial loading time of web applications developed using *Mascherl* [90].

The JavaScript library of *Mascherl* has the following dependencies to external JavaScript libraries:

- *history.js* - for handling the browser history stack.
- *jQuery* - required by history.js, plus heavily used for implementing AJAX communication and partial page updates.

Listing 3.1 from the previous chapter shows an example of the main static HTML page used by the prototype of *Mascherl*. In the head section of the HTML markup it includes *jQuery*, *bootstrap*, *history.js*, and finally also *Mascherl's* JavaScript library. The *bootstrap* library is thereby not required by *Mascherl*, however, it is used to ease layouting. Using this static HTML page for the initial full page request results in the web browser loading this HTML page among all these JavaScript libraries synchronously from the web server, before starting to render the web page, because the scripts are all placed in the head section of the HTML. Thus, the user has to wait until everything is loaded, before anything is displayed in the browser. To circumvent this behaviour, it is not possible to simply move all scripts from the head section to the HTML body section, which allows the browser to start rendering even before the scripts are loaded, because

this can cause severe race conditions, if the scripts are not loaded in the correct order [115]. These race conditions can, however, be bypassed by using *Require.js*.

In order to use *Require.js*, the JavaScript of *Mascherl* needs to be wrapped into a function by using AMD's *define()* method. Listing 4.12 shows the wrapper around the actual JavaScript code of *Mascherl*.

```
1  define('mascherl', ['jquery', 'history'], function($, History) {
2
3    var mascherl = {};
4
5    mascherl.navigate = function (url, container, page) { ... }
6
7    // other functions cut for clarity
8
9    return mascherl;
10 }
```

Listing 4.12: The JavaScript of *Mascherl* wrapped into AMD's *define()* function, in order to use *Require.js*. The two dependencies, *jQuery* and *history.js*, are supplied as parameters to the wrapper function, as opposed to using global JavaScript objects.

In addition, a configuration file for *Require.js* needs to be created, which is shown in Listing 4.13.

```
1  require.config({
2    paths: {
3      mascherl: '/mascherl/1.0.0/js/mascherl',
4      jquery: '/webjars/jquery/2.1.3/jquery.min',
5      history: '/webjars/historyjs/1.8.0/scripts/bundled/html5/jquery.history',
6      bootstrap: '/webjars/bootstrap/3.3.4/js/bootstrap.min'
7    },
8    shim: {
9      history: {
10       deps: ['jquery'],
11       exports: 'History'
12     },
13     bootstrap: {
14       deps: ['jquery']
15     }
16   },
17   deps: ['mascherl']
18 });
```

Listing 4.13: *Require.js* configuration file for *Mascherl* named *main.js*. The file specifies the paths to all modules, i.e. *mascherl*, *jquery*, *history*, and *bootstrap*, and configures the dependencies and exports of the non-AMD modules. Finally, the module *mascherl* is defined as the main module to be loaded.

Using the above configuration file, the static HTML page of *Mascherl* can use *Require.js* in order to load all JavaScript dependencies asynchronously and without race conditions in the body section of the HTML, which speeds up the rendering process of full page requests. The modified static HTML template is shown in Listing 4.14.

```
1  <html>
2  <head>
3      <meta charset="utf-8">
4      <title>{{title}}</title>
5      <link rel="stylesheet"
6            href="/webjars/bootstrap/3.3.4/css/bootstrap.min.css" />
7
8      <script>
9          window.mascherlConfig = {
10             applicationVersion: "{{applicationVersion}}",
11             pageGroup: "{{pageGroup}}"
12         };
13     </script>
14 </head>
15 <body>
16 <script src="/webjars/requirejs/2.1.17/require.min.js"></script>
17 <script src="/resources/1.0.0/main.js"></script>
18
19 {{$main}}{{/main}}
20
21 <script id="error" type="text/x-error-template"> ... </script>
22 </body>
23 </html>
```

Listing 4.14: The new version of the static HTML page of *Mascherl*, which uses *Require.js* and the configuration file *main.js* shown in Listing 4.13 in order to load the client-side libraries of the web application. As a result of using *Require.js*, the *script* tags can be moved to the HTML body section, which allows the browser to load the scripts asynchronously, and therefore speeds up rendering of the web site.

Both listings also show that *Mascherl* makes use of *WebJars* [141], which is a novel approach for managing client-side dependencies in JVM-based web applications, comparable to *Bower* for *node.js* [18]. The idea is to package JavaScript, CSS, and other client-side resources in JAR files, which are available in Maven repositories, and can therefore be referenced by the web application simply by adding it to the project descriptor of the web application, instead of manually adding the resources to the web archive. This effectively allows the dependency manager of the application (e.g. *Maven*) to handle client-side libraries in addition to server-side dependencies. This approach works without further configuration in Servlet 3 environments, because Servlet 3 containers serve resources located in subdirectories of *META-INF/resources* of every JAR on the web application classpath using the *default* servlet [74].

For production environments, *Require.js* also provides an optimizer and obfuscator, which effectively copies all JavaScript resources required by the web application on the initial load into one file that is then minimized and obfuscated. Instead of asynchronously loading the various different JavaScript files needed by the application using separate HTTP requests, this technique allows to serve all necessary JavaScript in only one HTTP request, which again speeds up the initial load of the application, because apart from fetching a smaller JavaScript file, the HTTP metadata is also only sent once, resulting in less traffic transferred between browser and web server. Additionally, only one TCP connection needs to be set up, which also saves connection

setup time. This optimization process can be triggered automatically upon building *Mascherl* with Maven by using the *Require.js* Maven plugin. For that to work, all JavaScript dependencies first have to be fetched from their *WebJars* archives, in order to be found by the optimizer. In addition, a configuration file is needed by the optimizer, which is shown in Listing 4.15.

```
1  ({
2    paths: {
3      mascherl: '${project.build.outputDirectory}/META-INF/resources/mascherl
           /1.0.0/js/mascherl',
4      jquery: '${project.build.directory}/webjars/META-INF/resources/webjars/
           jquery/2.1.3/jquery.min',
5      history: '${project.build.directory}/webjars/META-INF/resources/webjars/
           historyjs/1.8.0/scripts/bundled/html5/jquery.history'
6    },
7    shim: {
8      history: {
9        deps: ['jquery'],
10       exports: 'History'
11     }
12   },
13   baseUrl: '${project.build.sourceDirectory}/../resources/META-INF/resources/
         mascherl/1.0.0/js',
14   deps: ['mascherl'],
15   wrapShim: false,
16   optimize: 'uglify2'
17 })
```

Listing 4.15: The configuration file of the optimizer of *Require.js* used to create a minimized, obfuscated JavaScript bundle for *Mascherl*, containing *Mascherl's* JavaScript file, *jQuery*, and *history.js*. The used optimizer strategy is *uglify2*.

The above optimizer configuration file includes *Mascherl's* JavaScript file (8 kb), an already minimized version of *jQuery* (84 kb), and the jQuery bundle of *history.js* (16 kb). These JavaScript sources are togehter 108 kb in size. In comparison, the minimized bundle file produced by the optimizer is 104 kb in size, which is a minor improvement of 4 kb. That is because the major part of the optimized bundle file is contributed by *jQuery*, which is already minimized. Nevertheless, due to the fact that one HTTP transfer is used instead of three, the loading time of the web application is improved.

4.8 JavaScript extension points

In order for target web applications to trigger custom JavaScript logic upon certain client-side processes of *Mascherl*, the client library of *Mascherl* has to provide extension points. Certainly, various extension points can be utilized by target web applications, however, the most important one surely is the process triggered after a partial response is received from the web server. In that case, *Mascherl* triggers a custom *jQuery* event called *mascherlresponse* on the current window object, which contains the new URI set in the browser location bar and the name of the container that has been received. Listing 4.16 shows the code that triggers the aforementioned event, as well as a code snippet that shows how to attach a handler to that event.

```
1  // trigger mascherlresponse
2  $(window).triggerHandler("mascherlresponse", {
3    container: container,
4    pageUrl: History.getShortUrl(History.getState().url)
5  });
6
7
8  // attach handler to mascherlresponse
9  $(window).bind("mascherlresponse", function(event, data) {
10   // handle event
11 });
```

Listing 4.16: The respective JavaScript code of *Mascherl* that triggers the custom event *mascherlresponse*, as well as a snippet that shows how to attach a handler to that event.

Currently, *Mascherl* only provides this one event, which is enough to implement the example web application presented in the following chapter. However, additional extension points can easily be added in the future, if required.

4.9 Conclusion

In this chapter, various shortcomings of the prototype of *Mascherl* presented in the previous chapter have been identified, and then addressed in order to create a production ready version of *Mascherl*, which can be used to build stateless, RESTful web applications with reactive, partial server-side processing on top of a Servlet 3 conform container. The server-side part of those web applications can further be implemented completely in an asynchronous, non-blocking manner by using Reactive Extensions or Java 8's *CompletableFuture* API, as proposed in this chapter. Additionally, the methods for implementing controller classes and for creating *Mustache* based templates have been elaborated, in order to ease web application development.

The subsequent chapter presents an example web application that is built using the version of *Mascherl* presented in this chapter.

CHAPTER 5

An example web application

In this chapter, an example web application, which is built using the production-ready version of *Mascherl* developed in the previous chapter, is presented. The web application is a mocked web mail application, which allows users to send and receive e-mails, however, no real SMTP server is involved. The reason for this is that for the frontend it makes no difference whether or not the backend uses a real SMTP server, and therefore the additional effort is left out on purpose. Consequently, e-mails can only be exchanged between known users of the web application. In order to sign up for a new user account, the web application provides a registration process, in which an e-mail address can freely be chosen. After successful registration, users have to login into the application, in order to send and receive mock e-mails. The example application thus presents a solution for handling authentication in *Mascherl*. Furthermore, certain parts of the example web application are developed using Reactive Extensions and asynchronous, non-blocking processing. The web mail application supports the following use cases:

- Sign up of a new user.
- Login of an existing user.
- Show paginated lists of received, sent, composed, and deleted e-mails, respectively.
- Read an e-mail.
- Move an e-mail to the trash.
- Permanently delete e-mails from trash.
- Send an e-mail to other users of the application.
- Receive an e-mail from other users of the application.
- Logout from the application.

Figures 5.1 to 5.6 show screenshots of the web mail application, in which some of the above use cases are illustrated.

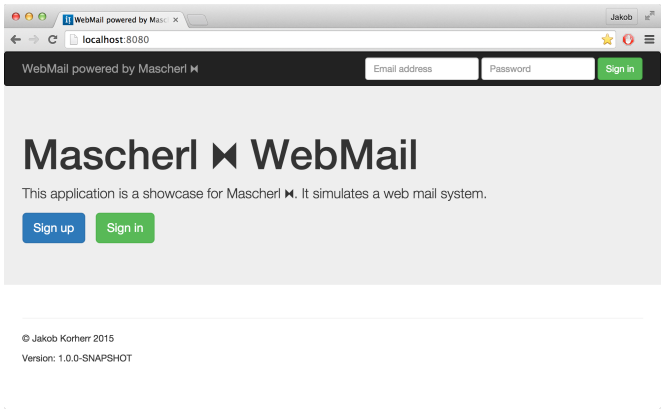
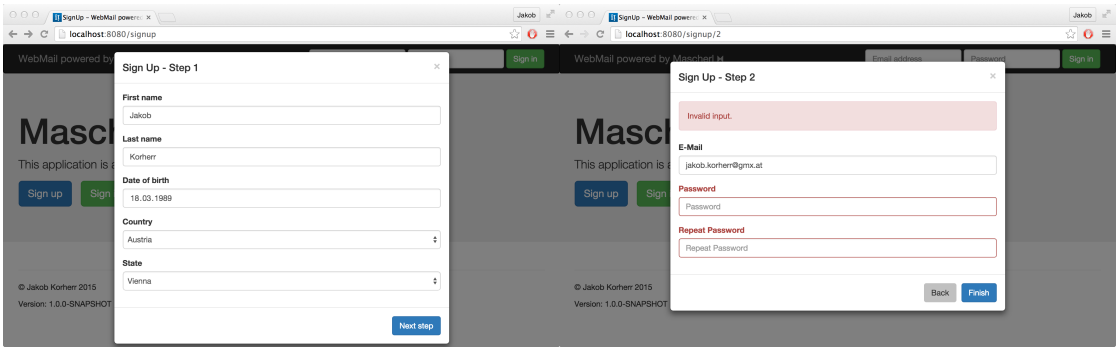
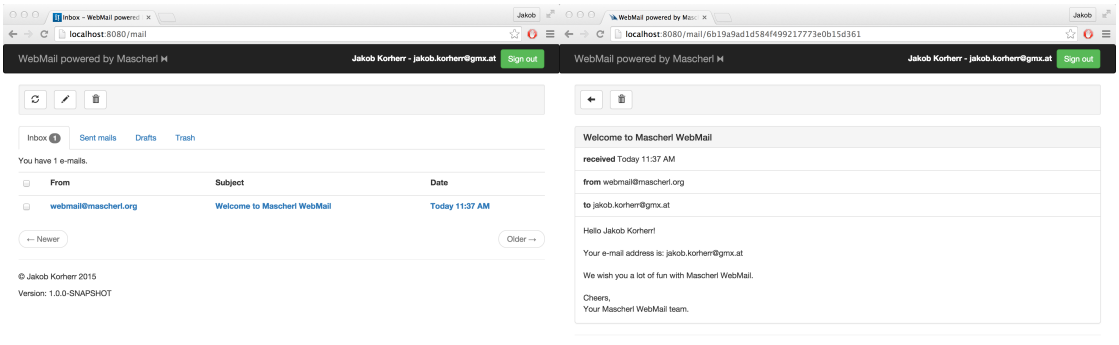


Figure 5.1: Root page of the web mail application. Allows new users to sign up, and existing users to sign in.



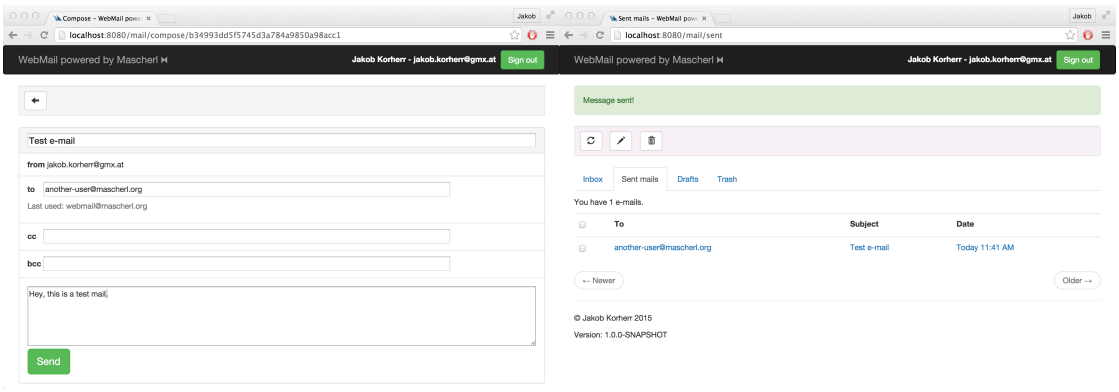
(a) Step one. (b) Step two, also showing validation errors.

Figure 5.2: Steps one and two of the sign up process of the web mail application. The whole sign up process is rendered in an own dialog.



(a) Inbox page shown after login. (b) Detail page of e-mail shown in inbox.

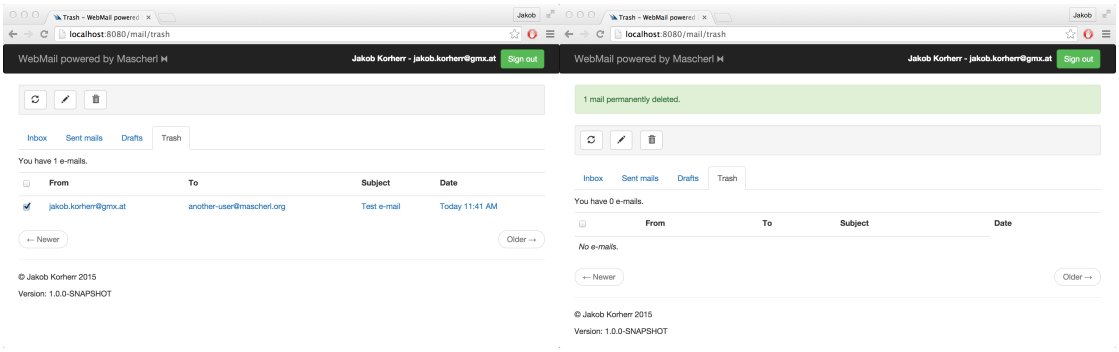
Figure 5.3: Inbox overview page and detail page of the e-mail shown in the inbox overview.



(a) E-mail composition page.

(b) Sent mails overview page.

Figure 5.4: Composition of a new e-mail, and the overview of sent e-mails including a success message, which is shown after the new e-mail has been successfully sent.



(a) Trash overview page with one e-mail.

(b) Trash overview page after e-mail has been deleted.

Figure 5.5: Permanent deletion of an e-mail in the trash overview page.

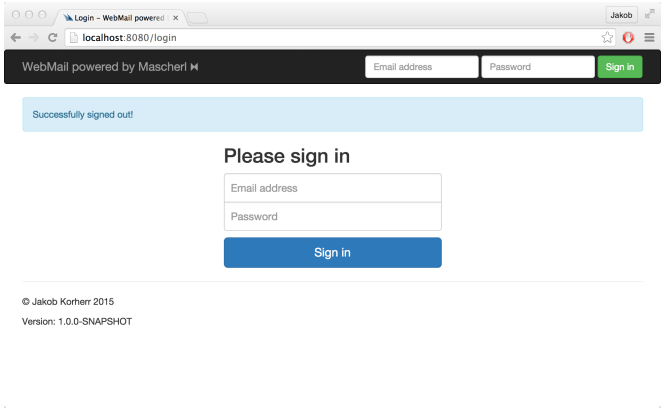


Figure 5.6: Logout page of the web mail application, including a sign out message.

5.1 Application setup

The web application project is Maven-based, and consists of the following major frameworks:

- *Mascherl 1.0.0* - Web UI layer.
- *Apache CXF 3.0.4* - JAX-RS implementation for *Mascherl*.
- *Spring 4.0.5* - Inversion of Control (IoC) container, bean manager, service layer.
- *Hibernate 4.3.8* - JPA persistence layer.
- *RxJava 1.0.9* - Reactive Extensions.
- *Hibernate Validator 5.2.0* - Bean Validation provider.
- *Bootstrap 3.3.4* - CSS framework.

The project is not divided into separate modules, but organized as one Java web archive (WAR). However, the code is split into three layers: user interface (UI) layer, service layer, and persistence layer. These layers are described in detail in the following.

5.2 Persistence layer

The persistence layer of the application uses the *Java Persistence API* (JPA) 2.1 as abstraction over the actual relational database system. As mentioned above, *Hibernate 4.3.8* is used as the underlying JPA implementation.

The JPA entity model is very simple, because it only consists of two entities:

- *UserEntity* - User attributes and login data.
- *MailEntity* - E-mails created, sent, or received by a user of the application.

Additionally, an abstract base entity is used in order to implement the primary key and version fields required for every entity. For the primary key of the above entities, UUIDs as generated by *java.util.UUID.randomUUID()* are used. In order to persist or delete these entities, the service layer directly uses the JPA *EntityManager*, whereas data queries are performed using the Java Persistence Query Language (JPQL).

To actually store the data, the following underlying database systems are used:

- *HSQLDB 2.3.2* - In-memory database.
- *PostgreSQL 9.4.1* - Persistent database.

HSQLDB is used for development and testing of the application, because it is a Java based in-memory database, which can be started and stopped automatically with the web application. However, the deployed version of the application uses PostgreSQL in order to persistently store the application data. Apart from that, PostgreSQL provides means for asynchronous, non-blocking communication with the database system, which is currently not available in HSQLDB.

5.3 Service layer

The service layer consists of several services, which are stateless and provide the application's core functionality to the UI layer. For the internal implementation every service may use other services, or access the persistence layer. This direct access to the persistence layer is not available in the UI layer, and thus the service layer separates the UI layer from the persistence layer. In order to enforce a strict separation, the service layer exclusively uses *domain* classes in the interfaces to the UI layer, i.e. for data input and output, whereas internally it uses the entities provided by the persistence layer. As a consequence, JPA classes never reach the UI layer.

Services are Java classes that are annotated with the Spring stereotype annotation *@Service*, which allows Spring to automatically discover all services in the application's classpath upon startup. After discovery, the services are instantiated and their dependencies, e.g. the JPA *EntityManager* of the persistence layer, are resolved. Afterwards, Spring makes the services available to the UI layer.

The web mail application uses the following services:

- *ComposeMailService* - Composing new e-mails, opening drafts for editing, and saving drafts.
- *LoginService* - Login of a user using e-mail address and password.
- *MailService* - Access an overview of a user's e-mails, get read and unread e-mail count, open an e-mail for reading, moving an e-mail to the trash, and permanently deleting an e-mail in the trash.
- *SendMailService* - Sending an e-mail.
- *SignUpService* - Sign up of a new user, and provide information for the sign up process (i.e. list of available countries and states).
- *TestDataService* - Insert test data upon application startup in development mode.

Most service implementations follow the same structure: At first the domain input data is verified and then transferred into the respective JPA data types, afterwards the persistence layer is called with this data, and finally the result of the persistence layer is transferred back into domain classes and then returned to the UI layer. Listing 5.1 shows an example service implementation of the web mail application.

```
1  @Service
2  public class ComposeMailService {
3
4    @PersistenceContext
5    private EntityManager em;
6
7    @Transactional
8    public Mail openDraft(String uuid, User currentUser) {
9      List<MailEntity> resultList = em.createQuery(
10         "select m " +
11             "from MailEntity m " +
```

```
12              "where m.uuid = :uuid " +
13              "and m.user.uuid = :userUuid", MailEntity.class)
14          .setParameter("uuid", uuid)
15          .setParameter("userUuid", currentUser.getUuid())
16          .setHint(QueryHints.HINT_READONLY, Boolean.TRUE)
17          .getResultList();
18      if (resultList.isEmpty()) {
19        return null;
20      }
21      MailEntity entity = resultList.get(0);
22      if (entity.getMailType() != MailType.DRAFT) {
23        throw new IllegalStateException(
24            "Mail with uuid " + uuid + " + is not of type draft");
25      }
26      return convertToDomain(entity);
27    }
28
29    // other methods cut for clarity
30  }
```

Listing 5.1: The service *ComposeMailService* of the web mail application with the implementation of *openDraft()*. The service uses the JPA *EntityManager* in order to access the persistence layer of the application. The service method queries for the referenced e-mail entity using JPQL, checks the state of the e-mail, and then converts the data to the domain class *Mail* and returns it to the caller.

Currently, JPA only supports synchronous invocations, because it builds upon JDBC, which at this time does not support asynchronous, non-blocking database interactions. As a consequence, the service layer can not be implemented in a non-blocking manner. However, the web mail application uses two ways to circumvent this restriction:

- Use Spring's task scheduling features in order to execute certain services on separate thread pools, and thus, make the service calls asynchronous from the perspective of the UI layer.
- Instead of using JPA, directly access the underlying database system using asynchronous, non-blocking database drivers (e.g. available for PostgreSQL and MySQL [11, 13]).

Listing 5.2 shows an asynchronous wrapper service over the synchronous *ComposeMailService* shown in Listing 5.1, which uses a separate task executor called *composeMailServiceExecutor* in order to make the service invocations asynchronous. The service wrapper returns and instance of *rx.Observable*, which can be used by the UI layer as shown in Listing 4.10.

```
1  @Service
2  public class ComposeMailServiceAsync {
3
4    @Inject
5    @Named("composeMailServiceExecutor")
6    private ThreadPoolTaskExecutor composeMailServiceExecutor;
7
```

```
8   @Inject
9   private ComposeMailService composeMailService;
10
11  public Observable<Mail> openDraft(String uuid, User currentUser) {
12    return Observable.<Mail>create((subscriber) -> {
13      subscriber.onStart();
14      try {
15        subscriber.onNext(composeMailService.openDraft(uuid, currentUser));
16      } catch (Throwable e) {
17        subscriber.onError(e);
18      }
19      subscriber.onCompleted();
20    }).subscribeOn(Schedulers.from(composeMailServiceExecutor));
21  }
22
23  // other methods cut for clarity
24 }
```

Listing 5.2: The asynchronous wrapper service over the synchronous *ComposeMailService*, which uses a separate task executor for the actual service invocations. The asynchronous callback mechanism is implemented using RxJava.

When running on the PostgreSQL database system, an asynchronous, non-blocking database driver for PostreSQL can be used instead of JPA, in order to implement asynchronous services without using separate task executors. Listing 5.3 shows the implementation of *getLastSendToAddressesAsync()*, which uses the asynchronous Java driver for PostgreSQL [13] and RxJava in order to implement a true asynchronous, non-blocking database access. However, the service implementation cannot use JPQL, and therefore has to use native SQL to query the database system.

```
1  @Service
2  public class ComposeMailService {
3
4    private ConnectionPool db;
5
6    @PostConstruct
7    public void init() {
8      db = new ConnectionPoolBuilder()
9          .hostname("localhost")
10         .port(5432)
11         .database("niotest")
12         .username("postgres")
13         .password("postgres")
14         .poolSize(20)
15         .build();
16   }
17
18   @PreDestroy
19   public void close() {
20     db.close();
21   }
22
```

```
23   public Observable<MailAddressUsage> getLastSendToAddressesAsync(
24       User currentUser, int limit) {
25     return Observable.<MailAddressUsage>create((subscriber) -> {
26       db.query("select distinct mto.address, m.datetime " +
27                   "from mail m " +
28                   "join mail_to mto on mto.mail_uuid = m.uuid " +
29                   "where m.user_uuid = $1 " +
30                   "and m.mail_type = $2 " +
31                   "and not exists (" +
32                   "   select 1 from mail m2 " +
33                   "   join mail_to mto2 on mto2.mail_uuid = m2.uuid " +
34                   "   where m2.user_uuid = $1 " +
35                   "   and m2.mail_type = $2 " +
36                   "   and mto2.address = mto.address " +
37                   "   and m2.datetime > m.datetime " +
38                   ") " +
39                   "order by m.datetime desc " +
40                   "limit $3",
41           Arrays.asList(currentUser.getUuid(), MailType.SENT.name(), limit),
42           result -> {
43             try {
44               subscriber.onStart();
45               TimestampColumnZonedDateTimeMapper dateTimeColumnMapper
46                   = new PersistentZonedDateTime().getColumnMapper();
47               StreamSupport.stream(result.spliterator(), false)
48                 .map(row ->
49                   new MailAddressUsage(
50                     new MailAddress(row.getString(0)),
51                     dateTimeColumnMapper.fromNonNullValue(
52                         row.getTimestamp(1))))
53                 .forEach(subscriber::onNext);
54               subscriber.onCompleted();
55             } catch (Exception e) {
56               subscriber.onError(e);
57             }
58           },
59           subscriber::onError);
60     });
61   }
62
63   // other methods cut for clarity
64 }
```

Listing 5.3: Asynchronous, non-blocking service implementation of *getLastSendToAddresses-Async()* using the asynchronous Java driver for PostgreSQL [13] in combination with RxJava. This approach does not require a separate task executor in order to provide an asynchronous service, but in return does not allow to use JPA, and therefore the database system has to be queried using native SQL, and the data conversion has to be performed manually.

5.4 User interface layer

The UI layer of the web mail application uses the production ready version of *Mascherl* presented in the previous chapter on top of *Apache CXF 3.0.4*. In addition, Spring is used to configure CXF, to manage the page classes, and to access the services provided by the service layer. Direct access to the persistence layer is prohibited.

The user interface is organized in the following different page classes, which correspond to the views shown in Figures 5.1 to 5.6:

- *IndexPage* - The root page of the web mail application, including login and logout actions.
- *MailComposePage* - The page for composing a new mail, including the actions for creating a new draft, saving a draft, opening a draft, and sending an e-mail.
- *MailDetailPage* - The page showing details of a specific e-mail, including the action for moving the e-mail to the trash.
- *MailInboxPage* - The inbox overview page, showing an overview of received, sent, created, and deleted e-mails, including the actions for moving marked e-mails to the trash, or for permanently deleting them.
- *SignUpPage* - The page for the sign up dialog of the application, including the actions for step one and two of the sign up process.

Listing 5.4 shows the Spring configuration for CXF and *Mascherl*, which includes the above page classes.

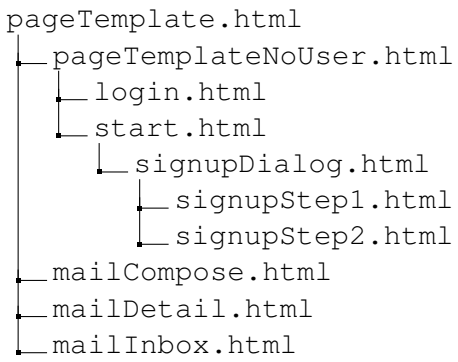
```
1  <beans> <!-- namespaces cut for clarity -->
2    <import resource="classpath:META-INF/cxf/cxf.xml" />
3    <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />
4    <import resource="classpath:META-INF/mascherl.spring.xml" />
5
6    <context:annotation-config />
7    <context:component-scan base-package="org.mascherl.example" />
8
9    <jaxrs:server address="/">
10     <jaxrs:serviceBeans>
11       <ref bean="indexPage" />
12       <ref bean="signUpPage" />
13       <ref bean="mailInboxPage" />
14       <ref bean="mailDetailPage" />
15       <ref bean="mailComposePage" />
16     </jaxrs:serviceBeans>
17     <jaxrs:providers>
18       <bean class="org.mascherl.jaxrs.MascherlMessageBodyWriter" />
19       <bean class="org.mascherl.jaxrs.MascherlRequestFilter" />
20       <bean class="org.mascherl.jaxrs.MascherlResponseFilter" />
21       <bean class="org.mascherl.example.filter.WebMailRequestFilter" />
22     </jaxrs:providers>
23     <jaxrs:inInterceptors>
24       <bean
```

```
25          class="org.mascherl.validation.cxf.CxfBeanValidationInInterceptor" />
26      </jaxrs:inInterceptors>
27      <jaxrs:invoker>
28        <bean class="org.mascherl.async.cxf.CxfObservableInvoker" />
29      </jaxrs:invoker>
30    </jaxrs:server>
31
32    <!-- JPA and service config cut for clarity -->
33  </beans>
```

Listing 5.4: Portion of the Spring configuration of the web mail application, which configures the CXF based JAX-RS server for *Mascherl*. The above page classes are added as service beans, *Mascherl's* extension point providers are registered, and the CXF specific Bean Validation interceptor as well as the Observable invoker are configured. In addition, the custom request filter *WebMailRequestFilter* is added to perform authentication checks for the web mail application.

Page templates

All page classes use associated *Mustache* template files for rendering, which extend the common main template *pageTemplate.html*. In addition, a static full page template is used, similar to the one shown in Listing 4.14. The following shows the template hierarchy of the application:



The template *pageTemplate.html*, which is shown in Listing 5.5, defines the main structure of the user interface and provides various sections, which can be overwritten by sub-templates, as presented in Section 4.2.

```
1  <nav class="navbar navbar-inverse">
2    <div class="container">
3      <div class="navbar-header">
4        <button type="button" class="navbar-toggle collapsed"
5                data-toggle="collapse"
6                data-target="#bs-example-navbar-collapse-1">
7          <span class="sr-only">Toggle navigation</span>
8          <span class="icon-bar"></span>
9          <span class="icon-bar"></span>
10         <span class="icon-bar"></span>
11       </button>
12
```

```
13        {{$titleBar}}
14          <a class="navbar-brand" href="/mail">
15              WebMail powered by Mascherl &#x29d3;
16          </a>
17        {{/titleBar}}
18      </div>
19      <p class="navbar-text navbar-right">
20        {{$userInfo}}
21          <form id="logoutFormNav" class="navbar-form navbar-right"
22                action="/logout" method="post">
23            <label style="color:white; margin-right: 5px;">
24              {{#user}}
25                <b>{{firstName}} {{lastName}} - {{email}}</b>
26              {{/user}}
27            </label>
28            <button type="submit" class="btn btn-success">Sign out</button>
29          </form>
30        {{/userInfo}}
31      </p>
32    </div>
33  </nav>
34  {{$headContainer}}
35  {{/headContainer}}
36  <div class="container">
37    {{$content}}
38      {{$messages}}
39        {{> messages}}
40      {{/messages}}
41      {{$pageContent}}
42      {{/pageContent}}
43    {{/content}}
44    <hr>
45    <footer>
46      <p>&copy; Jakob Korherr 2015</p>
47      <p>Version: {{applicationVersion}}</p>
48    </footer>
49  </div>
```

Listing 5.5: The main template file of the web mail application *pageTemplate.html*. Every other template of the application either directly extends this template, or extends a sub-template of this template as shown in the hierarchy above. The template defines the replaceable sections *titleBar*, *userInfo*, *headContainer*, *content*, *messages*, and *pageContent*.

Most other templates overwrite the sections *pageContent* or *headContainer* of the main template file in order to add actual content to the page. The template *pageTemplateNoUser.html*, however, overwrites the section *userInfo*, in order to be used by all sub-templates that are available without an authenticated user. Listing 5.6 shows the file *pageTemplateNoUser.html*.

```
1  {{< pageTemplate}}
2
3  {{$titleBar}}
4    <a class="navbar-brand" href="/">WebMail powered by Mascherl &#x29d3;</a>
5  {{/titleBar}}
```

```
{{$userInfo}}
  <form id="loginFormNav" class="navbar-form navbar-right" action="/login"
      method="post">
    <div class="form-group">
      <input name="email" type="email" placeholder="Email address"
             class="form-control" required>
    </div>
    <div class="form-group">
      <input name="password" type="password" placeholder="Password"
             class="form-control" required>
    </div>
    <button type="submit" class="btn btn-success">Sign in</button>
  </form>
{{/userInfo}}

{{/pageTemplate}}
```

Listing 5.6: The template file *pageTemplateNoUser.html*, which extends the main template of the application, and aims as the root template for all publicly available pages of the web application. The template overwrites the sections *titleBar* and *userInfo* of the main template. The new *titleBar* section contains a different home path of the application, and the overwritten *userInfo* section contains a login form instead of information about the current user.

The above template is used by the root page of the web mail application shown in Figure 5.1, the sign up dialog shown in Figure 5.2, and the login/logout page shown in Figure 5.6. The template of the root page is shown in Listing 5.7.

```
{{< ../pageTemplateNoUser}}

{{$headContainer}}
  <div class="jumbotron">
    <div class="container">
      <h1>Mascherl &#x29d3; WebMail</h1>
      <p>This application is a showcase for Mascherl &#x29d3;. It simulates a
          web mail system.</p>
      <p>
        <a class="btn btn-primary btn-lg" href="/signup"
           role="button" style="margin-right:10px;">Sign up</a>
        <a class="btn btn-success btn-lg" href="/login"
           role="button">Sign in</a>
      </p>
    </div>
  </div>
{{/headContainer}}

{{/ ../pageTemplateNoUser}}
```

Listing 5.7: The page template file of the root page of the web mail application called *start.html*. The template overwrites the section *headContainer* of the main template file in order to insert the content of the page. A screenshot of the rendered template is shown in Figure 5.1.

In contrast, all templates of pages, which are only available for authenticated users, directly extend the main template as shown in the template hierarchy above. Listing 5.8 shows the template file *mailDetail.html*, which is used by the detail page of an e-mail as shown in Figure 5.3 (b).

```
{{< ../pageTemplate}}
{{$pageContent}}
{{^mail}}
  <div class="alert alert-danger" role="alert">Mail not found.</div>
{{/mail}}
<div class="well well-sm">
  <button type="button" class="btn btn-default" aria-label="Go Back"
          onclick="history.back()" style="margin-right: 10px;">
    <span class="glyphicon glyphicon-arrow-left" aria-hidden="true"></span>
  </button>
  {{#mail}}
    <button id="deleteBtn" type="button" class="btn btn-default"
            aria-label="Delete" data-uuid="{{uuid}}">
      <span class="glyphicon glyphicon-trash" aria-hidden="true"></span>
    </button>
  {{/mail}}
</div>
{{#mail}}
  <div class="panel panel-default">
    <div class="panel-heading">
      <h3 class="panel-title">{{subject}}</h3>
    </div>
    <ul class="list-group">
      <li class="list-group-item"><b>{{dateTimeLabel}}</b> {{dateTime}}</li>
      <li class="list-group-item"><b>from</b> {{from}}</li>
      <li class="list-group-item"><b>to</b> {{to}}</li>
      {{#cc}}<li class="list-group-item"><b>cc</b> {{.}}</li>{{/cc}}
      {{#bcc}}<li class="list-group-item"><b>cc</b> {{.}}</li>{{/bcc}}
    </ul>
    <div class="panel-body">
      {{{messageText}}}
    </div>
  </div>
{{/mail}}
<script>
  require(['mailDetail'], function(mailDetail) {
    mailDetail.pageContentLoaded();
  });
</script>
{{/pageContent}}
{{/ ../pageTemplate}}
```

Listing 5.8: The page template file of the detail page of an e-mail called *mailDetail.html*. The template overwrites the section *pageContent* of the main template in order to insert the content of the page. A screenshot of the rendered template is shown in Figure 5.3 (b).

The page classes listed in the beginning of this section use all of the above templates in order to produce the desired HTML output. Table 5.1 gives an overview of supported URIs, the associated page classes, and the used templates or performed actions.

URI	Method	Page class	Template/Action
/	GET	IndexPage	start.html
/login	GET	IndexPage	login.html
/login	POST	IndexPage	Perform login of user
/logout	POST	IndexPage	Perform logout of user
/signup	GET	SignUpPage	signupStep1.html
/signup/2	GET	SignUpPage	signupStep2.html
/signup/selectCountry	POST	SignUpPage	Switch country in dialog
/signup/step1	POST	SignUpPage	Finish sign up step 1
/signup/step2	POST	SignUpPage	Finish sign up step 2
/mail	GET	MailInboxPage	mailInbox.html
/mail/sent	GET	MailInboxPage	mailInbox.html
/mail/draft	GET	MailInboxPage	mailInbox.html
/mail/trash	GET	MailInboxPage	mailInbox.html
/mail/delete	POST	MailInboxPage	Delete selected mails
/mail/{mailUuid}	GET	MailDetailPage	mailDetail.html
/mail/{mailUuid}/delete	POST	MailDetailPage	Delete referenced mail
/mail/compose/{mailUuid}	GET	MailComposePage	mailCompose.html
/mail/compose	POST	MailComposePage	Compose a new mail
/mail/send/{mailUuid}	POST	MailComposePage	Send an e-mail
/mail/save/{mailUuid}	POST	MailComposePage	Save a draft e-mail

Table 5.1: Overview of all supported URIs and HTTP request methods of the web mail application. Every combination of URI and HTTP method is associated with a respective Java method in a page class. For GET requests, this method renders the given template, whereas for POST requests, the associated method performs the specified action.

Page classes

The page classes of the web mail application are designed as presented in Section 4.3. The instances of the page classes and their dependencies to other page classes and to the service layer are managed by Spring. Each page class is therefore annotated with the Spring annotation *@Component*, and the properties referencing other page classes and services are annotated with *@Inject*. Every Java method, which corresponds to a specific HTTP request path and method, is annotated with the respective JAX-RS annotations, i.e. *@GET* or *@POST*, as well as *@Path*. All methods that handle GET requests return an instance of *MascherlPage*, whereas all methods that handle POST requests return an instance of *MascherlAction*. Listing 5.9 shows the page class *IndexPage* as an example for most page classes of the application. This class is used for the root, login, and logout pages of the web mail application, including all methods for rendering pages and executing actions.

```
1 @Component
2 public class IndexPage {
3
```

```
4   @Inject
5   private ValidationResult validationResult;
6
7   @Inject
8   private MascherlSession session;
9
10  @Inject
11  private LoginService loginService;
12
13  @Inject
14  private MailInboxPage mailOverviewPage;
15
16  @GET
17  @Path("/")
18  public MascherlPage start() {
19    return Mascherl.page("/templates/root/start.html")
20        .pageTitle("WebMail powered by Mascherl");
21  }
22
23  @GET
24  @Path("/login")
25  public MascherlPage login() {
26    return Mascherl.page("/templates/root/login.html")
27        .pageTitle("Login - WebMail powered by Mascherl");
28  }
29
30  @POST
31  @Path("/login")
32  public MascherlAction loginAction(@Valid @BeanParam LoginBean loginBean) {
33    User user;
34    if (validationResult.isValid()) {
35      user = loginService.login(loginBean.getEmail(),
36                                loginBean.getPassword());
37    } else {
38      user = null;
39    }
40    if (user != null) {
41      session.put("user", user);
42      return Mascherl
43          .navigate("/mail")
44          .renderAll()
45          .withPageDef(mailOverviewPage.inbox(1))
46          .withPageGroup("MailInboxPage");
47    } else {
48      return Mascherl
49          .navigate("/login")
50          .renderAll()
51          .withPageDef(login().container("messages", (model) ->
52              model.put("errorMsg", "Invalid email or password!")));
53    }
54  }
55
56
```

```
57    @POST
58    @Path("/logout")
59    public MascherlAction logoutAction() {
60        session.remove("user");
61        return Mascherl
62                .navigate("/login")
63                .renderAll()
64                .withPageDef(login().container("messages", (model) ->
65                    model.put("infoMsg", "Successfully signed out!")));
66
67    }
68 }
```

Listing 5.9: The page class *IndexPage*, which renders the root, login, and logout pages of the application, and handles the associated actions. The class accesses the *ValidationResult* as well as the *MascherlSession* objects of *Mascherl* via dependency injection provided by Spring. Furthermore, a reference to the page class *MailInboxPage* and to the service *LoginService* are injected. The page class contains two methods for rendering pages, i.e. the root page and the login/logout page of the application. Additionally, the class contains two methods for handling the login and the logout action, respectively.

The methods *start()* and *login()* of the above page class simply render the Mustache templates *start.html* and *login.html*, respectively, and set appropriate page titles. The action methods *loginAction()* and *logoutAction()* are more complex. The login action first checks the user input via Bean Validation, and then uses the *LoginService* to obtain a valid user object. If the credentials entered by the user are valid and the service returns a user object, this user object is stored in the *MascherlSession* and a navigation to the mail overview page is performed. Otherwise, the login page is rendered with an appropriate error message. The logout action removes the user object from the session and then renders the login page with an information message.
The UI layer of the application uses its own data model for reading body data from a POST request, and for providing data to the rendering engine. However, the service layer uses a different data model, i.e. the domain model of the application. Therefore, all data input that is forwarded to the service layer needs to be converted to the respective domain classes, and everything that is retrieved from the service layer needs to be converted to the appropriate data transfer objects used in the rendering engine. This principle is already manifested in the login action of the above listing. In the code, the instance of *LoginBean* is not forwarded directly to the service layer, but rather the individual properties, i.e. e-mail and password, are added separately. This concept becomes more apparent in other page classes, e.g. *MailDetailPage* and *MailComposePage*.
In the class *MailDetailPage* an e-mail is read from the service layer, which returns the domain class *Mail*. Instead of directly forwarding this domain class to the rendering engine, the page class first converts it into the data transfer object (DTO) *MailDetailDto*. This DTO only contains simple data fields (e.g. strings), which can be directly rendered by the rendering engine without further conversion or formatting.
The page class *MailComposePage* on the other hand receives data from the user in the form of the class *ComposeMailBean*. This bean class contains all input data necessary for sending a new e-mail, as well as the associated Bean Validation constraints. The data properties in the

bean class are, however, only native strings or numbers, and are thus not conform to the domain model used in the *SendMailService*. Therefore, the page class first needs to convert the native input data into the domain class *Mail*, before calling the service for sending the mail.
Using these separate data models in the UI and service layers enforces the separation of UI logic and business logic, and thus promotes a clean modularization of the project. Moreover, it supports the MVC push approach used by *Mascherl*.

Asynchronous controller methods

The page class shown in Listing 5.9 does not use *Mascherl's* asynchronous request processing features, but rather handles all HTTP requests synchronously. The reason for this is that most services of the service layer are synchronous, because the underlying JPA implementation is also synchronous. However, the application also provides some asynchronous services, which allow the usage of asynchronous HTTP request handling. One example is the page class *MailComposePage*, which is used for composing and sending new e-mails. The method *compose()*, which is used for rendering the compose page of a specific draft e-mail, is shown in Listing 5.10.

```
1  @GET
2  @Path("/mail/compose/{mailUuid}")
3  public Observable<MascherlPage> compose(
4      @PathParam("mailUuid") String mailUuid) {
5    User localUser = MascherlSession.getInstance().get("user", User.class);
6    Observable<List<MailAddressUsage>> sendToAddressesObservable =
7      composeMailService.getLastSendToAddressesAsync(
8          localUser, RECEIVER_HINT_MAX_ADDRESSES).toList()
9        .timeout(500, TimeUnit.MILLISECONDS,
10         Observable.just(Collections.emptyList()))
11       .onErrorReturn((throwable) -> Collections.emptyList());
12   Observable<List<MailAddressUsage>> receivedAddressesObservable =
13     composeMailServiceAsync.getLastReceivedFromAddresses(
14         localUser, RECEIVER_HINT_MAX_ADDRESSES)
15       .timeout(500, TimeUnit.MILLISECONDS,
16         Observable.just(Collections.emptyList()))
17       .onErrorReturn((throwable) -> Collections.emptyList());
18   return sendToAddressesObservable
19     .zipWith(
20       receivedAddressesObservable,
21       (sendToList, receivedFromList) -> {
22         List<MailAddressUsage> addresses
23             = new ArrayList<>(RECEIVER_HINT_MAX_ADDRESSES * 2);
24         if (receivedFromList != null) {
25           addresses.addAll(receivedFromList);
26         }
27         if (sendToList != null) {
28           addresses.addAll(sendToList);
29         }
30         return addresses.stream()
31             .distinct()
32             .sorted((u1, u2) -> u2.getDateTime().compareTo(u1.getDateTime()))
33             .limit(RECEIVER_HINT_MAX_ADDRESSES)
34             .collect(Collectors.toList());
```

```
35        }
36      )
37      .zipWith(
38        composeMailServiceAsync.openDraft(mailUuid, localUser),
39        (List<MailAddressUsage> receiverHintList, Mail mail) ->
40          Mascherl.page("/templates/mail/mailCompose.html")
41              .pageTitle("Compose - WebMail powered by Mascherl")
42              .container("userInfo", (model) -> model.put("user", localUser))
43              .container("pageContent", (model) -> {
44                if (mail != null) {
45                  model.put("mail", convertToPageModelForEdit(mail));
46                }
47                String receiverHint = receiverHintList.stream()
48                    .map((usage) -> usage.getMailAddress().getAddress())
49                    .collect(Collectors.joining(", "));
50                if (!receiverHint.isEmpty()) {
51                  model.put("receiverHint", receiverHint);
52                }
53              })
54      )
55      .onErrorReturn((throwable) -> {
56        if (throwable instanceof IllegalStateException) {
57          return Mascherl.deferredPage(() ->
58              mailDetailPage.mailDetail(mailUuid)
59                .replaceUrl(UriBuilder.fromMethod(
60                  MailDetailPage.class, "mailDetail").build(mailUuid))
61                .pageGroup("MailDetailPage"));
62        }
63        throw (RuntimeException) throwable;
64      });
65  }
```

Listing 5.10: The method *compose()* of the page class *MailComposePage*, which uses Reactive Extensions and three asynchronous services from the service layer in order to render the page for composing a specific draft e-mail asynchronously. All three service calls are combined using *zipWith()* in order to calculate the respective *rx.Observable* of *MascherlPage*. In addition, two service calls have timeouts with a default value, for the case that the service calls take too long.

Since the above method returns an instance of *rx.Observable* of *MascherlPage*, *Mascherl* handles the associated HTTP request asynchronously, as described in Section 4.6. This *rx.Observable* is the result of three asynchronous service calls, which are invoked and combined using Reactive Extensions. Figures 5.7 and 5.8 illustrate this sophisticated asynchronous process performed by the method *compose()*. If every service responds within its defined timeout, the process shown in Figure 5.7 applies. However, if the service call to *getLastSendToAddresses()* takes more than the defined timeout of 500 milliseconds, the process shown in Figure 5.8 applies.
This asynchronous request processing has many advantages: In the first place, the servlet container thread is not blocked for the duration of the database calls, and can therefore serve other clients in the meantime. Secondly, all three database calls can be processed simultaneously by the database system, instead of sequentially, which speeds up the overall service process. Finally, it is very easy to specify an upper bound for the response time of the HTTP request by

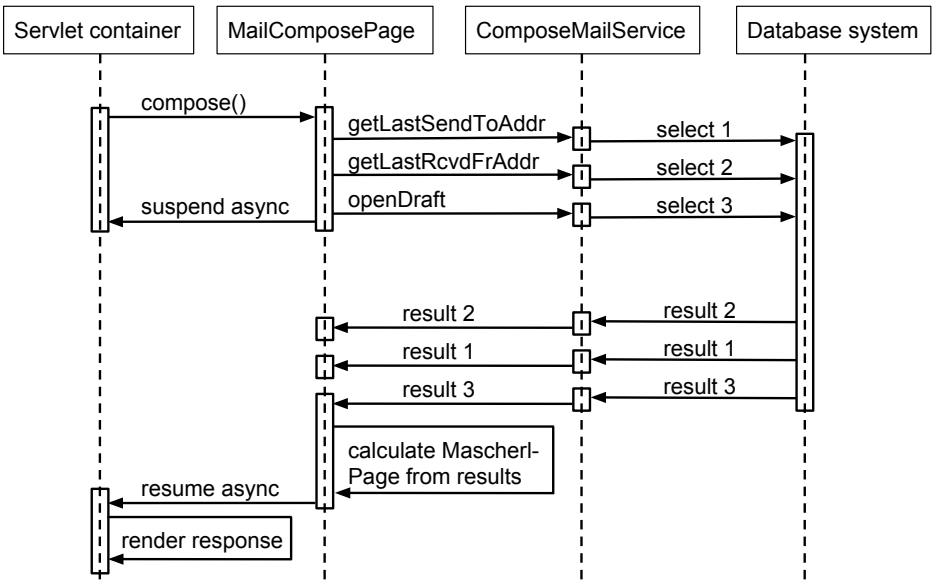


Figure 5.7: Sequence diagram of the asynchronous process performed by the method *compose()* in the page class *MailComposePage*. The method triggers three asynchronous service calls and then suspends the asynchronous request processing on the servlet container, and therefore frees the servlet container thread. If all three service calls respond within their defined timeouts, the results are combined and the respective *MascherlPage* is calculated. Finally, the asynchronous request processing of the servlet container is resumed and the response is rendered.

setting appropriate timeouts for all asynchronous service calls. If some services are unresponsive, the user still gets feedback from the web server in an acceptable time, rather than waiting for a very long period of time for the HTTP response.

Authentication management

As mentioned before, the web mail application provides publicly available web pages, as well as pages, for which an authenticated user is required. To implement this authentication management mechanism, the application uses the JAX-RS request filter *WebMailRequestFilter*, as referenced in Listing 5.4. This request filter is called before an incoming HTTP request is forwarded to the associated page class.
The application also uses the domain class *User* as shown in Listing 5.9 in order to store the current user of the application in the session of *Mascherl*. The user object is added to the session upon signing in, and is removed again from the session after logging out of the application.
For a simple differentiation between public and private pages, the application uses a special URI design, as shown in Table 5.1. All private pages start with the prefix */mail*, whereas all public pages do not share a common prefix. Thus, the request filter can easily determine if the current page is private, and in that case check if a valid user is available in the *MascherlSession*. If not, the filter can restrict further processing, and redirect the request to the login page of the

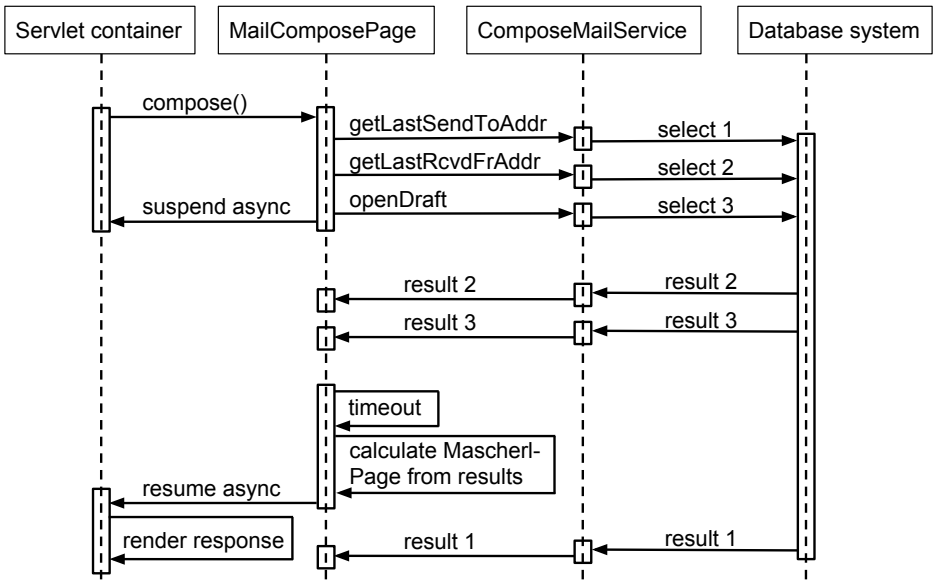


Figure 5.8: Sequence diagram of the same asynchronous process as shown in Figure 5.7, however, with the service call to *getLastSendToAddresses()* taking more than the defined timeout of 500 milliseconds. In that case a timeout handler is triggered in the page class, and the respective *MascherlPage* is calculated without the information from the unresponsive service. When the late result from the service is finally received, it is ignored.

application. Listing 5.11 shows the implementation of the authentication request filter.

```
1  @Priority(Priorities.AUTHENTICATION)
2  public class WebMailRequestFilter implements ContainerRequestFilter {
3
4    @Override
5    public void filter(ContainerRequestContext requestContext)
6        throws IOException {
7      MascherlSession session = MascherlSession.getInstance();
8      User user = session.get("user", User.class);
9      if (requestContext.getUriInfo().getPath().startsWith("mail")) {
10       if (user == null) {
11         if (isAjaxRequest(requestContext)) {
12           if (isGetRequest(requestContext)) {
13             requestContext.abortWith(Response.ok(
14                 Mascherl.navigate("/").redirect()).build());
15           } else {
16             requestContext.abortWith(Response.status(
17                 Response.Status.UNAUTHORIZED).build());
18           }
19         } else {
20           requestContext.abortWith(Response.seeOther(
21               UriBuilder.fromUri("/").build()).build());
22         }
```

```
23        }
24      } else {
25        if (user != null) {
26          if (isAjaxRequest(requestContext)) {
27            if (isGetRequest(requestContext)) {
28              requestContext.abortWith(Response.ok(
29                  Mascherl.navigate("/mail").redirect()).build());
30            }
31          } else {
32            requestContext.abortWith(Response.seeOther(
33                UriBuilder.fromUri("/mail").build()).build());
34          }
35        }
36      }
37    }
38
39    private boolean isGetRequest(ContainerRequestContext requestContext) {
40      return Objects.equals("GET", requestContext.getRequest().getMethod());
41    }
42
43    private boolean isAjaxRequest(ContainerRequestContext requestContext) {
44      return Objects.equals(
45          requestContext.getHeaderString("X-Requested-With"),
46          "XMLHttpRequest");
47    }
48
49  }
```

Listing 5.11: The JAX-RS request filter used for user authentication in the web mail application. If the current request path starts with */mail*, it checks for a valid user in the session of *Mascherl*, and if there is none, it redirects the request to the login page of the application using the appropriate redirect variant. For POST requests, the HTTP status code *401 Unauthorized* is returned. Furthermore, the filter also checks if an authenticated user accesses one of the public pages of the application, and in that case redirects the user to the authenticated area. This behaviour is not required from an authentication point of view, but desired for the web mail application.

Data input and form validation

As mentioned before, the web mail application makes use of Bean Validation to validate any kind of user input. Listing 5.9 above already shows how *Mascherl's ValidationResult* is used to verify the login credentials of a user. Even though this sample shows how Bean Validation is integrated into a *Mascherl* based action method, it is not comprehensive. Therefore, the following demonstrates how Bean Validation is utilized in the sign up dialog of the application (shown in Figures 5.2 (a) and (b)).
The Mustache based template of the dialog contains HTML input fields for every desired data property. Those input fields are decorated with a certain CSS error class, if a validation error for the field exists, which is manifested by the existence of a corresponding property in the page

model, e.g. *firstNameError*. An excerpt of the whole page template is shown in Listing 5.12, showing the input fields for *firstName* and *lastName*.

```
{{< signupDialog}}
{{$dialogContent}}
{{#bean}}
<form id="signUpForm" action="/signup/step1" method="post">
  <!-- other elements cut for clarity -->

  <div class="form-group{{#firstNameError}} has-error{{/firstNameError}}">
    <label for="inputFirstName" class="control-label">First name</label>
    <input id="inputFirstName" type="text" name="firstName"
      class="form-control" placeholder="First name"
      value="{{firstName}}" required autofocus>
  </div>

  <div class="form-group{{#lastNameError}} has-error{{/lastNameError}}">
    <label for="inputLastName" class="control-label">Last name</label>
    <input id="inputLastName" type="text" name="lastName"
      class="form-control" placeholder="Last name"
      value="{{lastName}}" required>
  </div>

  <!-- other elements cut for clarity -->
</form>
{{/bean}}
{{/dialogContent}}
{{/ signupDialog}}
```

Listing 5.12: A part of the Mustache based page template *signupStep1.html*, which includes the HTML markup for the input fields for first name and last name of the user, who wants to sign up for a new web mail account.

The underlying Java bean, which is populated with the input data provided by the user, is shown in Listing 5.13. Using the *@BeanParam* annotation on the corresponding action method instructs the JAX-RS framework to automatically create an instance of this class and to fill it with the available form data. The mapping from HTML form input fields to Java bean properties is specified using *@FormParam* annotations, which reference the respective HTML input element names. Moreover, the properties also hold the validation constraints for Bean Validation, using the associated annotations, e.g. *@Size* or *@NotNull*.

```
public class SignUpStep1Bean {

    @FormParam("firstName")
    @Size(min = 1)
    @NotNull
    private String firstName;

    @FormParam("lastName")
    @Size(min = 1)
    @NotNull
    private String lastName;
```

```
12
13      @FormParam("dateOfBirth")
14      @NotNull
15      @Pattern(regexp = "[0-9]{4}-[0-9]{2}-[0-9]{2}")
16      private String dateOfBirth;
17
18      @Past
19      @NotNull
20      @JsonIgnore
21      public LocalDate getDateOfBirthParsed() {
22          try {
23              return LocalDate.parse(dateOfBirth, DateTimeFormatter.ofPattern("
                    yyyy-MM-dd"));
24          } catch (RuntimeException e) {
25              return null;
26          }
27      }
28
29      @FormParam("country")
30      @Size(min = 2)
31      @NotNull
32      private String country;
33
34      @FormParam("state")
35      @Size(min = 1)
36      @NotNull
37      private String state;
38
39      // getters and setters cut for clarity
40
41  }
```

Listing 5.13: The Java bean class *SignUpStep1Bean*, which holds the data provided by a user in the first step of the sign up process of the web mail application. The data is stored in the properties of the class, which are also annotated with respective Bean Validation annotations, enforcing certain constraints on the properties.

Finally, the page class using the above template and Java bean is shown in Listing 5.14. As mentioned above, the mapping from HTML form input parameters to the associated Java bean is triggered by using *@BeanParam* on the parameter type of the action method. Bean Validation of the respective Java bean is initiated by using the *@Valid* annotation in addition to *@BeanParam*. The result of the Bean Validation, including possible validation errors, is exposed to the action method using the class *ValidationResult*. The action method decides its outcome based on the *ValidationResult*. If the validation succeeded, the data is stored in the session and the next dialog step is shown. Otherwise, the current dialog step is re-rendered, including an error message and the error markers for the input fields, which have validation errors.

```
1  @Component
2  public class SignUpPage {
3
4    @Inject
5    private SignUpService signUpService;
```

```
6
7   @Inject
8   private ValidationResult validationResult;
9
10  @Inject
11  private MascherlSession session;
12
13  @Inject
14  private IndexPage indexPage;
15
16  @POST
17  @Path("/signup/step1")
18  public MascherlAction signUpStep1(@Valid @BeanParam SignUpStep1Bean bean) {
19    if (validationResult.isValid()) {
20      session.put("signUpStep1", bean);
21      return Mascherl
22          .navigate("/signup/2")
23          .renderContainer("dialogContent")
24          .withPageDef(signUpStep2());
25    } else {
26      return Mascherl
27          .stay()
28          .renderContainer("dialogContent")
29          .withPageDef(signUp()
30              .container("dialogContent", (model) -> {
31                  model.put("bean", bean);
32                  model.put("countries",  convertToSelectOptions(
33                      signUpService.getCountries(), bean.getCountry()));
34                  addValidationErrors(model,
35                      "firstName", "lastName",
36                      "dateOfBirth", "country", "state");
37                  if (hasValidationError("dateOfBirthParsed")) {
38                      model.put("dateOfBirthError", true);
39                  }
40              })
41              .container("stateContainer", (model) -> model.put("states",
42                  convertToSelectOptions(signUpService.getStates(
43                      bean.getCountry()), bean.getState())))
44              .container("dialogMessages", (model) -> model.put("errorMsg",
45                  "Invalid input.")));
46    }
47  }
48
49  private void addValidationErrors(Model model, String... fields) {
50    Arrays.stream(fields)
51        .filter(this::hasValidationError)
52        .forEach(field -> model.put(field + "Error", true));
53  }
54
55  private boolean hasValidationError(String field) {
56    return validationResult.getConstraintViolations().stream()
57        .map(ConstraintViolation::getPropertyPath)
58        .map((path) -> {
```

```
59            Node last = null;
60            for (Node node : path) {
61              last = node;
62            }
63            return last == null ? null : last.getName();
64          })
65          .anyMatch((property) -> Objects.equals(property, field));
66    }
67
68    // other methods cut for clarity
69
70  }
```

Listing 5.14: A part of the page class *SignUpPage*, including the action method for the submission of the first step of the sign up dialog of the web mail application. The action method uses automatic form input conversion into a Java bean, as well as Bean Validation of the data using the *@Valid* annotation. The outcome of the action method is decided by the result of Bean Validation.

JavaScript of the application

The web mail application integrates with the *Require.js* based approach for handling JavaScript provided by *Mascherl*. Therefore, the application provides separate *Require.js* modules for every web page of the application. The different modules are:

- mailCompose.js
- mailDetail.js
- mailInbox.js
- webmail.js

The file *webmail.js* contains the main module of the application. It is used to configure and boot *Mascherl's* JavaScript modules. The three other files contain the JavaScript application logic for their associated web pages. Listing 5.8 shows the reference of *mailDetail.js* in the template file *mailDetail.html* using *Require.js*.
All of these module files are combined with the JavaScript modules of *Mascherl* into one optimized and minified JavaScript file, which is loaded once for the whole application using the same mechanism as presented in Section 4.7.

Development modus

In a production environment *Mascherl* parses each template file exactly once, and then caches the parsed data structure representing the template file in memory for faster processing. This means that on the first request to a web page on a newly deployed web application, the *Mustache* based rendering engine reads the associated template file on the file system, and then parses it, in order to render the response. However, every subsequent request to the same web page on the

same web application server uses the cached version of the parsed template file, which has been automatically stored in memory in the *Mustache* template cache. This cache is never evicted, which means that no template file is read twice, as long as the application is not re-deployed.
This behaviour makes sense in a production environment, because it can dramatically improve the rendering process. However, while the web application is still under development, this mechanism prevents hot deployments of page templates from taking effect on the server. This means that every time a page template is changed during development, the web application server has to be restarted in order for the changes to be picked up by *Mascherl*. To circumvent this restriction, *Mascherl* provides a development modus, which has been heavily utilized in the development of the web mail application. The development modus can be activated in the application configuration using `org.mascherl.developmentModus=true`.

5.5 Conclusion

In this chapter an example single-page web application has been developed, which uses *Mascherl* for the user interface layer, whereas the backend is implemented using established Java technologies. The application applies partial page rendering with back- and forward-button support, while entirely using bookmarkable URIs for alls its web pages, as summarized in Table 5.1. The chapter presents a complete application architecture, as well as a structure for page templates, and page classes. Moreover, the usage of asynchronous, non-blocking request handling has been presented, and solutions for implementing authentication management, user input, and form validation have been suggested. Finally, by adhering to the stateless architecture of *Mascherl*, this example web application can be scaled horizontally without server synchronization.
The following chapter uses this web application for evaluating the performance of *Mascherl*.

CHAPTER 6

Evaluation

In this chapter the performance of *Mascherl* is evaluated using the example web application presented in the previous chapter. First, *Mascherl's* partial page rendering approach is checked against traditional full page processing by comparing the response times and response sizes of a defined sequence of pages and actions in the example web application. Afterwards, the behaviour of the application serving various numbers of concurrent requests is probed, comparing traditional synchronous, blocking request handling with asynchronous, non-blocking request processing as promoted by *Mascherl*.

6.1 Evaluation setup

For the sake of automation and repeatability, Apache JMeter 2.13 (r1665067) [9] is used to carry out the load and performance testing. This tool allows to capture a sequence of HTTP requests, and then to replay it, while behaving like a normal web browser with regards to HTTP header and cookie management, however, not actually rendering the response or running any kind of JavaScript. Instead, the tool records the response time and size for every HTTP request, calculates some additional metrics, and finally allows to export all of this data. In the process, the tool allows to adjust the number of concurrent threads and the iteration count.
The tests are conducted on an Apple MacBook Pro Retina, 13-inch, Early 2013 with a 2.6 GHz Intel Core i5 processor and 8 GB 1600 MHz DDR3 main memory running OS X 10.9.5. The web application is hosted on an Apache Tomcat 8.0.15 server, which runs on a Java HotSpot 64-Bit Server VM (build 25.20-b05).

6.2 Full page versus partial page requests

In order to ascertain the benefits of *Mascherl's* partial page rendering approach over traditional full page processing, a complete workflow in the example web application is executed using Apache JMeter. For the evaluation of partial page rendering, the same HTTP requests are trig-

gered by JMeter, which would be triggered by a normal user running through the same workflow in a web browser. For full page requests, however, the additional HTTP header parameter *X-Mascherl-Force-Full-Page* has been added to *Mascherl*, which allows to force full page processing, regardless of any other parameters or any rendering logic of the application. Adding this HTTP header parameter to all HTTP requests in JMeter allows to get accurate results for traditional full page rendering without modifying the underlying *Mascherl* based web application.
The performance tests have been run through the following workflow 50 times with 20 concurrent threads, resulting in 1000 probes for every particular step in the workflow:

1. go to start page (initial request, always full page)
2. show login page
3. incorrect login attempt
4. successfull login and show inbox
5. open mail from inbox
6. show inbox
7. show sent mails
8. show draft mails
9. show mails in trash
10. show inbox
11. compose new mail
12. send mail
13. show mail
14. delete mail
15. show mails in trahs
16. show mail
17. permanently delete mail
18. show inbox
19. logout

The comparison of average response times between full page and partial page request handling is shown in Figure 6.1. The figure shows that the response times of partial requests are always lower than the response times of full page requests, which can easily be explained by the additional model evaluation and the additional rendering efforts of full page requests. Thus, *Mascherl's* partial page processing approach allows faster browsing through the same web application, compared to traditional full page processing. Concretely, in the above workflow partial page rendering is on average 11.6% faster than full page processing.

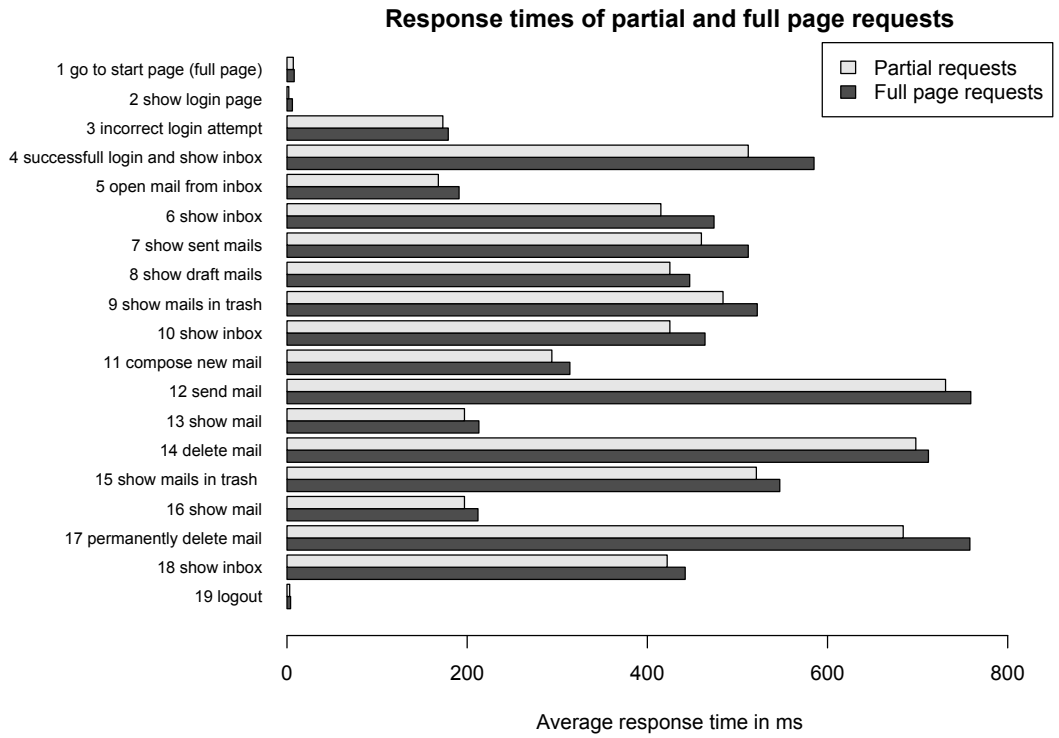


Figure 6.1: Bar plot comparing the average response times in milliseconds of *Mascherl's* partial page processing approach with traditional full page rendering in a predefined workflow of the example web application developed in the previous chapter.

The superiority of *Mascherl's* partial page processing approach becomes more apparent when comparing the average response sizes with traditional full page request handling. Figure 6.2 shows the respective bar plot. Full page processing always renders the whole HTML page and sends it to the client, therefore completely replacing the current page shown in the web browser. *Mascherl's* partial page rendering approach, however, uses the existing page in the browser, and only renders those parts of the page, which are actually changed, plus some additional metadata. The average reduction in transferred bytes in the above workflow is 2043 bytes, or 29.1%. This comparison already takes into account that the first request to the web application is always a full page request, which can easily be deduced from the bar plot in Figure 6.2. Furthermore, the bar plot shows that the same use case causes different amounts of transferred data in partial

rendering mode, depending on the previous page in the workflow. For example, the use case *show inbox* needs more data in step 6 of the workflow than in step 10, because the necessary changes to the respective previous pages are smaller in the transition from step 9 to 10, than from step 5 to 6.

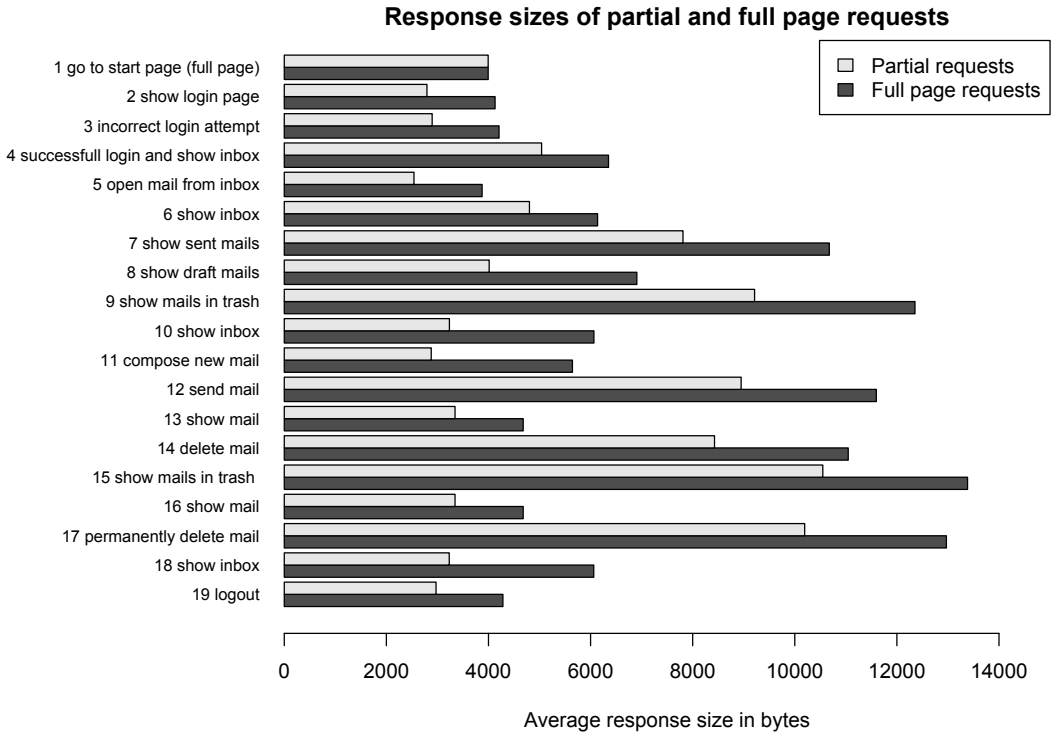


Figure 6.2: Bar plot comparing the average response sizes in bytes of *Mascherl's* partial page processing approach with traditional full page rendering in a predefined workflow of the example web application developed in the previous chapter.

Considering the big reduction of response sizes of 29.1% in the example workflow, an even greater speedup of response times than the measured 11.6% is to be expected for a real world web application, which is served over the internet, because the above performance test has been run on the local machine, and thus without taking network latency into account.

6.3 Synchronous versus asynchronous request handling

Apart from introducing partial page rendering to reduce response times and sizes, *Mascherl* also promotes the usage of asynchronous, non-blocking request handling, as opposed to synchronous, blocking request processing found in many Java EE applications. As already discussed in earlier chapters, asynchronous processing does not occupy one server thread for the whole processing of one client request, but rather uses one thread to process the request until the first call to a backend system is performed, and then uses another thread to process the result of that backend call as

soon as it is available. Therefore one client request can occupy multiple server threads, but not at the same time, and most importantly, not for performing blocking calls, which would effectively block the whole server thread. Consequently, a synchronous web server needs more server threads in order to handle the same number of concurrent client requests as an asynchronous web server.
In order to evaluate the performance margin between synchronous and asynchronous request handling in the *Mascherl* based web application presented in the previous chapter, the asynchronous action described in Section 5.4 and shown in Listing 5.10 is used. For evaluation purposes, a synchronous version of the same method has been developed, which produces the exact same output as the asynchronous version, however, invoking all three backend calls synchronously and in a blocking manner. In addition, the timeout restrictions of the backend calls of the asynchronous version of the method have been removed for the evaluation.

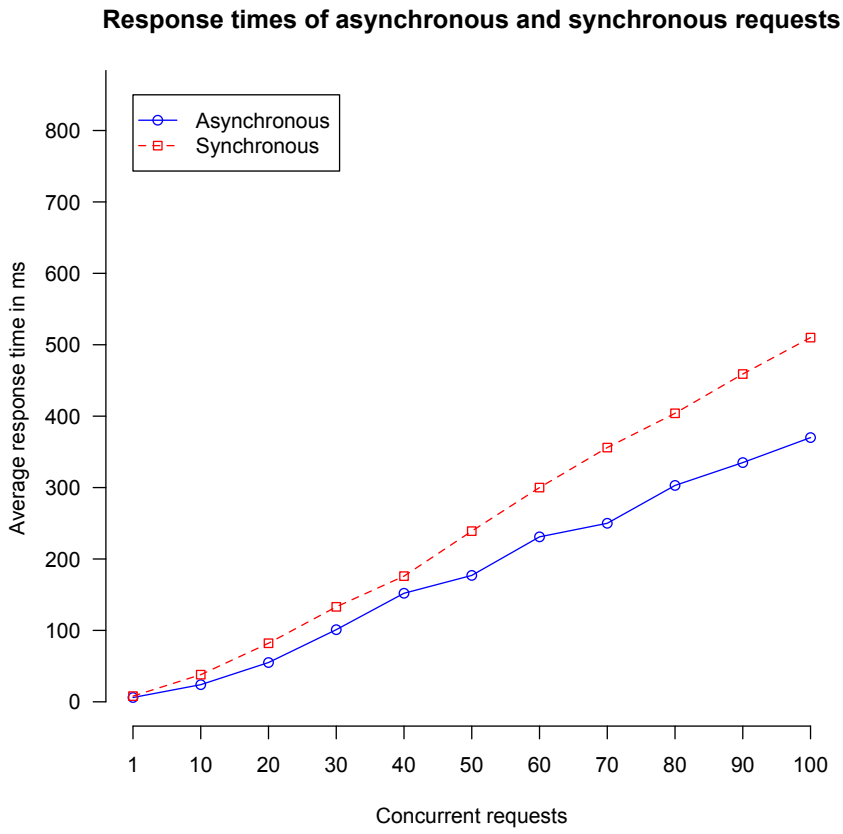


Figure 6.3: Chart showing the average response times in milliseconds with different numbers of concurrent requests (1 to 100), where each of the concurrent requests has been repeated 50 times in order to calculate a robust average value. The red line indicates synchronous request handling, whereas the blue line indicates asynchronous request processing.

Figure 6.3 compares the average response times of 50 consecutive HTTP calls per thread to the compose page of the example web application developed in the previous chapter with different numbers of concurrent threads (from 1 to 100) using asynchronous request handling on the one hand, and synchronous request handling on the other hand. It can be seen that synchronous request handling is less performant compared to asynchronous request processing. The main reason for this is that asynchronous processing allows to perform multiple simultaneous backend calls (as e.g. shown in Figure 5.7), whereas synchronous request processing only allows consecutive backend invocations. In practice, most web requests need more than one backend call to be processed, thus enabling parallel processing capabilities, which can be exploited best when using asynchronous processing methods. Therefore, a request that triggers three backend calls has been deliberately chosen for this evaluation.

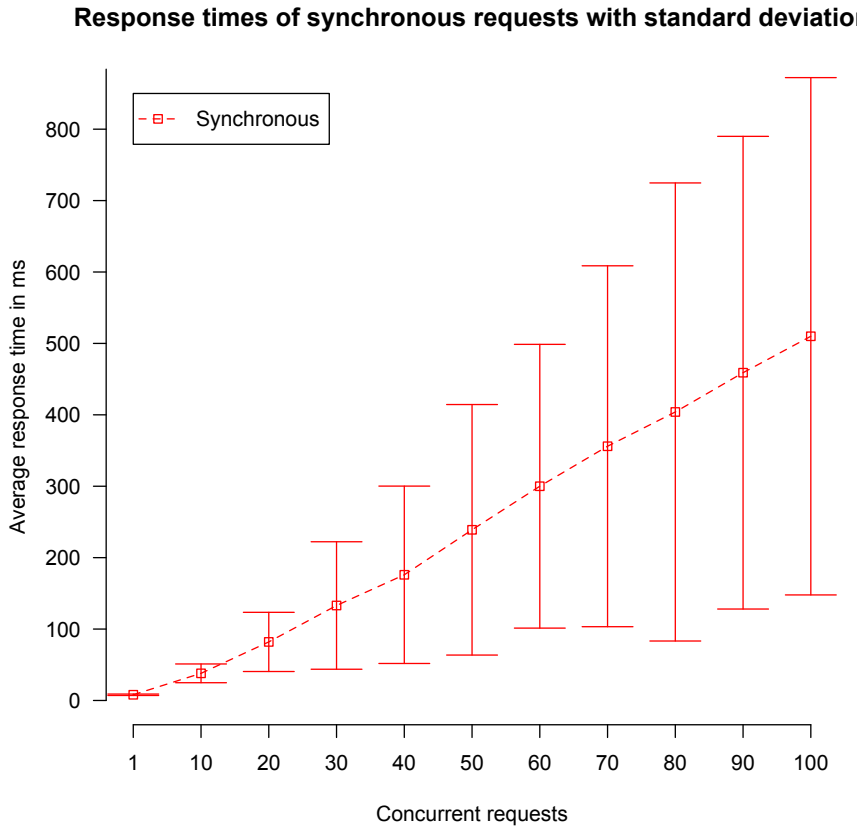


Figure 6.4: Synchronous part of the graph from Figure 6.3, including the standard deviation of the results. The plot shows a high standard deviation in response times.

However, much more interesting is the fact that the asynchronous graph grows slower than the synchronous one, meaning that asynchronous request handling allows to serve more concurrent client requests than synchronous request processing in the same time, and consequently asyn-

chronous processing scales better than its synchronous counterpart.
The graph in Figure 6.3 does not include the standard deviation of the average response times, which is, however, very significant. Therefore, Figures 6.4 and 6.5 show the graphs of average response times including the standard deviation for synchronous and asynchronous request handling, respectively.

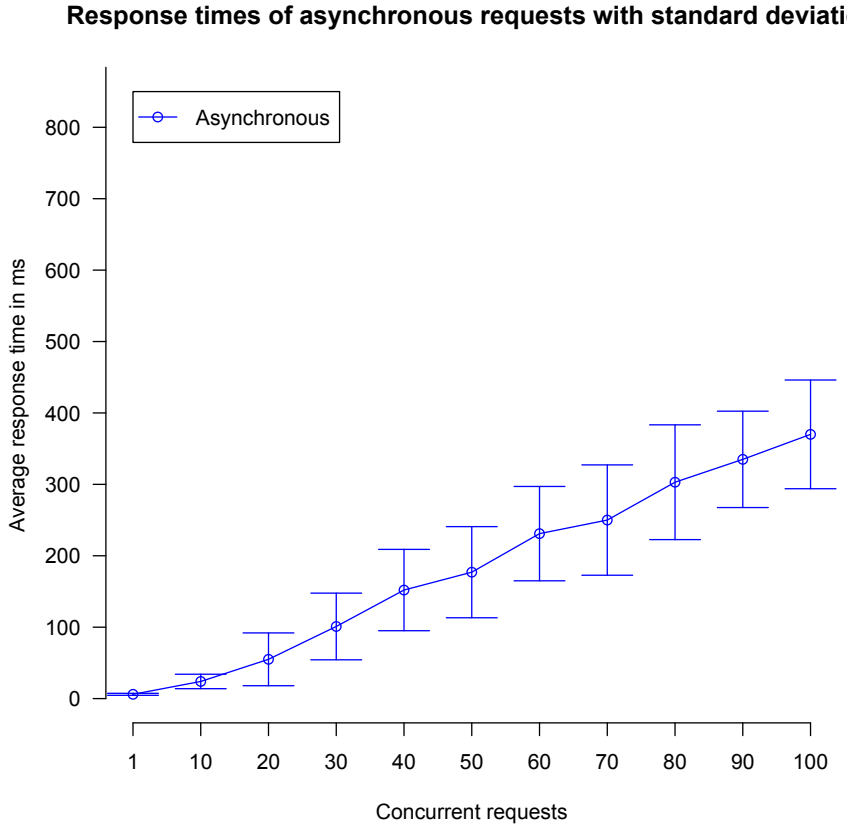


Figure 6.5: Asynchronous part of the graph from Figure 6.3, including the standard deviation of the results. The plot shows only a low standard deviation in response times.

Although the average response times of both approaches are very similar for 1 to 100 concurrent requests, the standard deviation is much higher in the synchronous case. This can be explained by the use of limited thread pools by the servlet container, i.e. Apache Tomcat. In the synchronous case, some HTTP requests immediately get a free thread, which processes this request until it is completed, possibly spending many processor cycles in a blocked state while waiting for backend calls to complete. Nevertheless, after the backend calls return, the same thread can instantly continue to process the request, and eventually complete it. These “lucky” requests are processed very fast, thus showing very low response times. Other “unlucky” requests do not get a free thread from the server’s thread pool right away, and thus need to wait for a free server

thread before the request processing can even begin. When they finally get picked up by a server thread, the actual request processing takes just as much time as the processing of the "lucky" requests, however, the total response time of the HTTP request is much higher, because of the initial waiting time. In the asynchronous case there are, of course, also "lucky" and "unlucky" requests, but nevertheless, the overall waiting times for free threads are much lower. That is, because in asynchronous processing mode no thread is blocked when performing backend calls, but rather starts processing another request in the meantime. However, when a backend call returns, it is not immediately processed further by the original thread, but instead has to wait for a free server thread just like a new HTTP request. Thus, waiting for a free server thread can happen multiple times in the processing of only one HTTP request, but the particular waiting times are much lower, because the individual server threads are never occupied for a long time. Altogether, asynchronous request handling is *fairer* in allocating processing resources than its synchronous counterpart.
Consequently, using synchronous request handling results in decreased predictability of application performance. Actual response times can at the same time be very low and quite high, depending on the resource scheduler of the web server. In the asynchronous graph, the standard deviation remains low even with a high number of concurrent requests.

Horizontal scalability

The above scalability evaluation has been run on only one web server, however, *Mascherl* seamlessly supports using multiple web servers that host the same web application, without the need for server synchronisation. That is because *Mascherl* based web applications are stateless by design, and consequently there is no state that needs to be synchronized between web servers. To achieve such a horizontal scalability of a *Mascherl* based web application, multiple web servers that host the same web application have to be installed, and additionally one or more load-balancers have to be put in place, which distribute the incoming HTTP requests over the available web servers. These load-balancers do not have to use the same web server for every request of the same client (i.e. sticky sessions), but can freely choose any web server for any incoming request. Thus, load-balancers can be kept very simple, which minimizes the overall impact on the request processing time.
Consequently, from the perspective of the frontend, a *Mascherl* based web application can be horizontally scaled indefinitely, as long as there are enough web servers available. However, the backend has to be scaled as well, otherwise the excellent scalability of the frontend is of no use.

6.4 Conclusion

The evaluations show that *Mascherl*'s partial page processing approach in combination with asynchronous request handling leads to smaller response sizes, faster response times, and a higher number of concurrent requests handled at the same time, compared to traditional Java EE web frameworks. Therefore, *Mascherl* based web applications are much more resource efficient than other Java EE based web applications. In addition, due to its statelessness, *Mascherl* allows its web applications to be distributed over various web servers, without the need for server com-

munication. Consequently, *Mascherl* enables building fast and highly scalable web applications on top of Java EE infrastructure.

CHAPTER 7

Conclusion and future work

7.1 Conclusion

The main contribution of this thesis is the novel web framework *Mascherl* along with an example web application that uses this framework for the implementation of its UI layer. *Mascherl* is the first Java based web framework with server-side rendering, which tightly integrates the HTML5 history API in its request handling approach. Previous efforts for using the HTML5 history API were only carried out by client-side JavaScript frameworks like AngularJS, or by external plugins for non-Java web frameworks like PJAX for Ruby on Rails. This tight integration of the HTML5 history API allows to keep the browser location bar in sync with the displayed web content, even though the framework exclusively uses AJAX requests for page updates. Therefore, *Mascherl* provides complete bookmarkability support, as well as a fully navigable browser history stack, i.e. the browser back and forward buttons work as expected.
The web framework *Mascherl* entirely uses partial page AJAX requests for any page updates, only the initial request to the web application is a full page request. Technically, the user never leaves the web page, which is rendered in the first request to the web application, because all subsequent pages are in fact only partial page updates to the respective previous pages. Consequently, *Mascherl* uses a single-page interface approach. Evaluations show that *Mascherl's* partial page processing mechanism uses 29.1% smaller HTTP responses and is on average 11.6% faster, compared to traditional full page rendering found in most Java EE frameworks.
For partial page rendering, *Mascherl* uses Mustache based HTML templates in combination with a MVC push based approach. Hence, the effective page model is calculated in the controller method, before the (partial) rendering process is triggered, leading to a deterministic model and predictable rendering times, as opposed to a MVC pull based approach where many model values are calculated on-demand during rendering. Furthermore, Mustache is logic-less, preventing any form of rendering logic to be placed into view templates, and therefore requiring all UI logic to be executed by the controller, putting the results in the model at disposal for the view templates. Consequently, a very strict separation of duties is enforced by *Mascherl*.

Mustache's novel template inheritance feature is used for specifying the page fragments of a template, which can be processed and rendered independently by *Mascherl*, thus no modification of the Mustache template syntax is necessary. On top of partial page rendering, *Mascherl* also applies partial model evaluation, in order to calculate only that part of the whole page model, which is needed to render the requested page fragment. For that, the API makes heavy use of lambda functions introduced in Java 8.
Apart from that, *Mascherl* promotes stateless servers by providing a custom session implementation, which gets along without any server state. Consequently, *Mascherl* based web applications can be scaled horizontally (i.e. by adding new web servers), without the need for web server synchronization. This enables easy installation in cloud based environments, i.e. IaaS and PaaS.
Finally, the usage of asynchronous, non-blocking request processing is endorsed by *Mascherl*. Two asynchronous API types are supported: Java 8's native *CompletableFuture* API, and the more elaborate *rx.Observable* API from RxJava. Using these APIs, it is possible to free the current processing thread of the servlet container while backend tasks are performed, and upon availability of the result(s), resuming the request processing on a different server thread. As a consequence, server threads are not occupied with blocking backend calls, but rather used to perform actual calculations, which usually are of short duration. The web server can therefore handle more concurrent requests with a much smaller thread pool, compared to synchronous request handling approaches, as demonstrated in the evaluation of *Mascherl*. Furthermore, many web pages contain the joint information of multiple backend calls, which can be fetched in parallel using asychronous processing, whereas synchronous methods only allow to call the respective backends in sequence. Thus, *Mascherl* allows to parallelize backend calculations for each request, resulting in even faster response times.

7.2 Future work

Even though *Mascherl's* suitability for state-of-the-art web application development has been demonstrated, there are still some points that can be elaborated. Definitions of pages and page actions can become confusing as soon as the page (action) gets complex, because of the heavy usage of lambda functions. The suitability and viability of *Mascherl's* API has to be observed, and future refinements will have to be applied.
One specific API, where future refinement is highly desired, is the interface for asynchronous page and action definitions. Currently, *Mascherl* only allows to provide the whole definition asynchronously, however, it is not possible to calculate the models of only some page sections (i.e. those that require a backend call) asynchronously.
In order to improve developer experience, IDE support for Mustache templates and *Mascherl's* container definitions are desirable future enhancements. Furthermore, logic-less templating languages like Mustache can in the future be the common basis of web designers and web application developers, lessening the dual efforts in UI development. That is because this kind of templating languages is easy to learn for web designers, and additionally, real data based rendering of templates can be simulated with JSON based data using the JavaScript implementation of the rendering engine.
Realtime communication with the web server via WebSockets or SockJS has been excluded from

this thesis as a non-target, because of the additional constraints that this feature imposes. However, modern web applications often make use of realtime communications, e.g. for chats, live data updates, or user collaboration. Hence, methods for integrating *Mascherl's* fully stateless processing model with such full-duplex communication channels are to be explored.
The client-side features of *Mascherl*, like link interception, AJAX request handling, and browser history stack management, depend highly on jQuery and history.js at the moment. *PJAX* [52] is an alternative implementation that provides similar client-side functionality, originally developed for Ruby on Rails based backends. Future work may detach *Mascherl's* built-in client-side library, make it exchangeable, and use other solutions like *PJAX* instead. Additionally, separate projects can emerge, which are solely concerned with the client-side part of partial page web frameworks that use a single-page interface in combination with the HTML5 history API.
Similarly, other server side frameworks (or plugins for existing web frameworks) may be launched as a result of the findings of *Mascherl*, which also employ real partial model evaluation and partial page rendering, in order to improve response times. Moreover, the novel state saving approach as well as the integration of different asynchronous APIs may be adopted by other web frameworks.
In general, it is highly desirable that more web frameworks start adopting

- the HTML5 history API,
- logic-less templating languages and the associated MVC-push approach,
- asynchronous request processing,
- and state saving mechanisms that promote stateless servers.

Bibliography

[1] Alexei White. Measuring the Benefits of Ajax . http://www.developer.com/xml/article.php/3554271/Measuring-the-Benefits-of-Ajax.htm. Accessed: 2015-04-22.

[2] AngularJS. https://www.angularjs.org/. Accessed: 2015-04-24.

[3] AngularJS API Reference, $locationProvider . https://docs.angularjs.org/api/ng/provider/$locationProvider. Accessed: 2015-04-26.

[4] AngularJS API Reference, ngRoute, $route . https://docs.angularjs.org/api/ngRoute/service/$route. Accessed: 2015-04-26.

[5] AngularJS Developer Guide, Templates. https://docs.angularjs.org/guide/templates. Accessed: 2015-05-28.

[6] AngularJS Developer Guide, Using $location. https://docs.angularjs.org/guide/$location. Accessed: 2015-04-23.

[7] Apache CXF: An Open-Source Services Framework. http://cxf.apache.org/. Accessed: 2015-05-29.

[8] Apache CXF, Bean Validation Feature. http://cxf.apache.org/docs/validationfeature.html. Accessed: 2015-06-04.

[9] Apache JMeter. http://jmeter.apache.org/. Accessed: 2015-08-01.

[10] Apache License, Version 2.0. http://www.apache.org/licenses/LICENSE-2.0. Accessed: 2015-04-24.

[11] Async, Netty based, database drivers for PostgreSQL and MySQL written in Scala. https://github.com/mauricio/postgresql-async. Accessed: 2015-06-23.

[12] Asynchronous Module Definition API specification. https://github.com/amdjs/amdjs-api/blob/master/AMD.md. Accessed: 2015-04-24.

[13] Asynchronous PostgreSQL Java driver. https://github.com/alaisi/postgres-async-driver. Accessed: 2015-06-23.

[14] Dian Ayuba, Amirah Ismail, and Mohd Isa Hamzah. Evaluation of Page Response Time between Partial and Full Rendering in a Web-based Catalog System. *Procedia Technology*, 11:807–814, 2013.

[15] A. Barth. HTTP State Management Mechanism. RFC 6265, RFC Editor, April 2011.

[16] Tim Berners-Lee, Roy Thomas Fielding, and L Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, RFC Editor, January 2005.

[17] Christos Bouras and Agisilaos Konidaris. Estimating and eliminating redundant data transfers over the web: a fragment based approach. *International Journal of Communication Systems*, 18(2):119–142, 2005.

[18] Bower: A package manager for the web. http://bower.io/. Accessed: 2015-06-07.

[19] Daniel Brodie, Amrish Gupta, and Weisong Shi. Accelerating dynamic Web content delivery using keyword-based fragment detection. In *Web Engineering*, pages 359–372. Springer, 2004.

[20] Can I use history? http://caniuse.com/#search=history. Accessed: 2015-04-25.

[21] Jim Challenger, Paul Dantzig, Arun Iyengar, and Karen Witting. A fragment-based approach for efficiently creating dynamic web content. *ACM Transactions on Internet Technology (TOIT)*, 5(2):359–389, 2005.

[22] Trieu C Chieu, Ajay Mohindra, Alexei A Karve, and Alla Segal. Dynamic scaling of web applications in a virtualized cloud computing environment. In *ICEBE'09. IEEE International Conference on e-Business Engineering, 2009*, pages 281–286. IEEE, 2009.

[23] Christopher L Merrill. Using Ajax to improve the bandwidth performance of web applications. http://www.webperformance.com/library/reports/ajax-bandwidth-testing/. Accessed: 2015-04-22.

[24] Jason Chu and Thomas R Dean. Automated Migration of List Based JSP Web Pages to AJAX. In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 217–226. IEEE, 2008.

[25] Dan Kegel. The C10K problem. http://www.kegel.com/c10k.html. Accessed: 2015-05-06.

[26] Dan Webb. It's About The Hashbangs . http://danwebb.net/2011/5/28/it-is-about-the-hashbangs. Accessed: 2015-04-23.

[27] Dive into dogs, An online example of the HTML5 history API. http://diveintohtml5.info/examples/history/fer.html. Accessed: 2015-04-25.

[28] Sadek Drobi. Play2: A New Era of Web Application Development. *Internet Computing, IEEE*, 16(4):89–94, 2012.

[29] Cristian Duda, Gianni Frey, Donald Kossmann, Reto Matter, and Chong Zhou. Ajax crawl: Making ajax applications searchable. In *IEEE 25th International Conference on Data Engineering (ICDE'09)*, pages 78–89. IEEE, 2009.

[30] EmberJS. http://emberjs.com/. Accessed: 2015-04-24.

[31] EmberJS API, Ember.Location. http://emberjs.com/api/classes/Ember.Location.html. Accessed: 2015-04-28.

[32] EmberJS Guide, Handlebars Basics . http://guides.emberjs.com/v1.10.0/templates/handlebars-basics/. Accessed: 2015-05-28.

[33] EmberJS Guide, Rendering a template . http://guides.emberjs.com/v1.10.0/routing/rendering-a-template/. Accessed: 2015-05-31.

[34] EmberJS Guide, Routing. http://guides.emberjs.com/v1.10.0/routing/. Accessed: 2015-04-27.

[35] Benjamin Erb. Concurrent Programming for Scalable Web Architectures. Master's thesis, Institute of Distributed Systems, Ulm University, 2012.

[36] Zhang Ermei, Liu Chen, and Yang Zhengqiu. Use Ajax to Prevent Repeat Form Submission. In *ISCSCT'08. International Symposium on Computer Science and Computational Technology, 2008*, volume 1, pages 261–264. IEEE, 2008.

[37] Bryan G Estrada. *Take Me Back: A Study of the Back Button in the Modern Internet*. PhD thesis, California Polytechnic State University San Luis Obispo, 2011.

[38] Facebook Engineering Blog, Using HTML5 Today . https://www.facebook.com/note.php?note_id=438532093919. Accessed: 2015-04-24.

[39] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455, RFC Editor, December 2011.

[40] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[41] Gil Fink and Ido Flatow. *Pro Single Page Application Development: Using Backbone.js and ASP.NET*. Apress, 1. edition, 2014.

[42] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 1 edition, 2002.

[43] Francisco J García, Raúl Izquierdo Castanedo, and Aquilino A Juan Fuente. A double-model approach to achieve effective model-view separation in template based web applications. In *Web Engineering*, pages 442–456. Springer, 2007.

[44] Jesse James Garrett et al. Ajax: A new approach to web applications. *Adaptive Path*, 2005.

[45] GitHub Engineering Blog, The Tree Slider. https://github.com/blog/760-the-tree-slider. Accessed: 2015-05-06.

[46] GitHub Gist, Template inheritance for Mustache. https://gist.github.com/spullara/1854699. Accessed: 2015-06-03.

[47] GitHub, gwt-pushstate. https://github.com/jbarop/gwt-pushstate. Accessed: 2015-04-29.

[48] GitHub, Jackson Project Home. https://github.com/FasterXML/jackson. Accessed: 2015-05-30.

[49] GitHub, Mustache Specification, Proposal: Template inheritance. https://github.com/mustache/spec/issues/38. Accessed: 2015-06-03.

[50] GitHub, Mustache.java. https://github.com/spullara/mustache.java. Accessed: 2015-05-28.

[51] GitHub, Ozark Samples. https://github.com/spericas/ozark/tree/master/test. Accessed: 2015-05-28.

[52] GitHub, pjax = pushState + ajax. https://github.com/defunkt/jquery-pjax. Accessed: 2015-05-16.

[53] GitHub, Play Framework sources, Crypto.scala . https://github.com/playframework/playframework/blob/master/framework/src/play/src/main/scala/play/api/libs/Crypto.scala. Accessed: 2015-05-30.

[54] GitHub, Play Framework sources, Http.scala . https://github.com/playframework/playframework/blob/master/framework/src/play/src/main/scala/play/api/mvc/Http.scala. Accessed: 2015-05-30.

[55] GitHub profile page of jakobk. https://github.com/jakobk?tab=repositories. Accessed: 2015-04-20.

[56] Google. Making AJAX Applications Crawlable - Full specification. https://developers.google.com/webmasters/ajax-crawling/docs/specification. Accessed: 2015-04-21.

[57] Google Plus, Managing Page State using HTML5 History. https://plus.google.com/+googleplus/posts/HisxUQ5XpKK. Accessed: 2015-05-06.

[58] Google Web Toolkit. http://www.gwtproject.org/. Accessed: 2015-05-06.

[59] Google Web Toolkit, Developer Guide, History Mechanism. http://www.gwtproject.org/doc/latest/DevGuideCodingBasicsHistory.html. Accessed: 2015-04-29.

[60] Google Web Toolkit, Javadoc of class com.google.gwt.user.client.History. http://www.gwtproject.org/javadoc/latest/com/google/gwt/user/client/History.html. Accessed: 2015-04-30.

[61] Brad Green and Shyam Seshadri. *AngularJS*. O'Reilly Media, Inc., 1 edition, 2013.

[62] Ilya Grigorik. Making the web faster with HTTP 2.0. *Communications of the ACM*, 56(12):42–49, 2013.

[63] Hao Han, Yinxing Xue, Oyama Keizo, and Yang Liu. Practice and evaluation of pagelet-based client-side rendering mechanism. *IEICE Transactions on Information and Systems*, 97(8):2067–2083, 2014.

[64] Cal Henderson. *Building scalable web sites*. O'Reilly Media, Inc., 1 edition, 2006.

[65] High Scalability Blog, Google+ Is Built Using Tools You Can Use Too: Closure, Java Servlets, JavaScript, BigTable, Colossus, Quick Turnaround. http://highscalability.com/blog/2011/7/12/google-is-built-using-tools-you-can-use-too-closure-java-ser.html. Accessed: 2015-05-28.

[66] history.js on GitHub. https://github.com/browserstate/history.js/. Accessed: 2015-04-25.

[67] history.js wiki on GitHub. Intelligent State Handling. https://github.com/browserstate/history.js/wiki/Intelligent-State-Handling. Accessed: 2015-04-23.

[68] HTML5 specification, Loading Web pages: History traversal. https://html.spec.whatwg.org/multipage/browsers.html#history-traversal. Accessed: 2015-04-24.

[69] HTML5 specification, Loading Web pages: Session history and navigation. https://html.spec.whatwg.org/multipage/browsers.html#history. Accessed: 2015-04-23.

[70] Htmleasy, A simple, elegant HTML page rendering web framework for Resteasy (JAX-RS). https://github.com/voodoodyne/htmleasy. Accessed: 2015-05-09.

[71] Ivar Grimstad, An update from JSR 371 (MVC 1.0). http://www.agilejava.eu/2015/01/25/an-update-from-jsr-371-mvc-1-0. Accessed: 2015-05-09.

[72] Ivar Grimstad, Samples for MVC 1.0 (JSR-371), HelloController.java. https://github.com/ivargrimstad/mvc-samples/blob/master/hello/src/main/java/eu/agilejava/mvc/HelloController.java. Accessed: 2015-05-14.

[73] Java Community Process, JSR 314: JavaServer Faces 2.0 Specification. https://jcp.org/en/jsr/detail?id=314. Accessed: 2015-05-08.

[74] Java Community Process, JSR 315: Java Servlet 3.0 Specification. https://jcp.org/en/jsr/detail?id=315. Accessed: 2015-05-06.

[75] Java Community Process, JSR 339: JAX-RS 2.0: The Java API for RESTful Web Services. https://jcp.org/en/jsr/detail?id=339. Accessed: 2015-05-11.

[76] Java Community Process, JSR 349: Bean Validation 1.1. https://jcp.org/en/jsr/detail?id=349. Accessed: 2015-06-04.

[77] Java Community Process, JSR 371: Model-View-Controller (MVC 1.0) Specification. https://jcp.org/en/jsr/detail?id=371. Accessed: 2015-05-09.

[78] Java Community Process, JSR 372: JavaServer Faces (JSF 2.3) Specification. https://jcp.org/en/jsr/detail?id=372. Accessed: 2015-04-30.

[79] Java.net, Model-View-Controller (MVC 1.0) Specification. https://java.net/projects/mvc-spec/pages/Home. Accessed: 2015-05-09.

[80] Java.net, Results from the Java EE 8 Community Survey. https://java.net/downloads/javaee-spec/JavaEE8_Community_Survey_Results.pdf. Accessed: 2015-05-09.

[81] Jersey - RESTful Web Services in Java. https://jersey.java.net/. Accessed: 2015-05-29.

[82] Joost Willemsen, Improving User Workflows with Single-Page User Interfaces. http://www.uxmatters.com/mt/archives/2006/11/improving-user-workflows-with-single-page-user-interfaces.php. Accessed: 2015-05-16.

[83] Jeremy Keith. The Future of DOM Scripting. In *DOM Scripting: Web Design with JavaScript and the Document Object Model*, pages 293–309. Springer, 2005.

[84] Jeremy Keith. Hijax: Progressive enhancement with ajax. *Proceedings of XTech, Amsterdam*, 2006.

[85] Federico Kereki. JavaScript all the way down. *Linux Journal*, 2015(250):1, 2015.

[86] Chulyun Kim and Kyuseok Shim. Text: Automatic template extraction from heterogeneous web pages. *IEEE Transactions on Knowledge and Data Engineering*, 23(4):612–626, 2011.

[87] Roland Kübert, Gregory Katsaros, and Tinghe Wang. A RESTful implementation of the WS-Agreement specification. In *Proceedings of the Second International Workshop on RESTful Design*, pages 67–72. ACM, 2011.

[88] Meng Li and Xiaohu Yang. An improved HTTP session management model in distributed environment. In *2010 2nd International Conference on Computer Engineering and Technology (ICCET)*, volume 6, pages 490–495. IEEE, 2010.

[89] LinkedIn Engineering Blog, The client-side templating throwdown: mustache, handlebars, dust.js, and more. https://engineering.linkedin.com/frontend/client-side-templating-throwdown-mustache-handlebars-dustjs-and-more. Accessed: 2015-05-28.

[90] Alex Liu. JavaScript and the Netflix user interface. *Communications of the ACM*, 57(11):53–59, 2014.

[91] Mark Pilgrim. Dive into HTML5: Manipulating history for fun & profit. http://diveintohtml5.info/history.html. Accessed: 2015-04-23.

[92] P Mazzetti, S Nativi, and L Bigagli. Integration of REST style and AJAX technologies to build Web applications; an example of framework for Location-Based-Services. In *ICTTA 2008. 3rd International Conference on Information and Communication Technologies: From Theory to Applications, 2008*, pages 1–6. IEEE, 2008.

[93] Erik Meijer. Reactive extensions (Rx): curing your asynchronous programming blues. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 11. ACM, 2010.

[94] Ali Mesbah. *Analysis and Testing of Ajax-based single-page web applications*. PhD thesis, Delft University of Technology, 2009.

[95] Ali Mesbah and Arie Van Deursen. An architectural style for Ajax. In *The Working IEEE/IFIP Conference on Software Architecture, 2007. WICSA'07.*, pages 9–9. IEEE, 2007.

[96] Ali Mesbah and Arie van Deursen. Migrating multi-page web applications to single-page Ajax interfaces. In *11th European Conference on Software Maintenance and Reengineering, 2007. CSMR'07.*, pages 181–190. IEEE, 2007.

[97] Microsoft Data Developer Center, Reactive Extensions for .NET. https://msdn.microsoft.com/en-us/data/gg577609. Accessed: 2015-05-08.

[98] Mozilla Developer Network, Manipulating the browser history. https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/Manipulating_the_browser_history. Accessed: 2015-04-24.

[99] San Murugesan. Understanding Web 2.0. *IT professional*, 9(4):34–41, 2007.

[100] Mustache. Logic-less templates. https://mustache.github.io/. Accessed: 2015-05-28.

[101] Den Odell. Using Client-Side Templates. In *Pro JavaScript Development*, pages 341–368. Springer, 2014.

[102] Jaewon Oh, Woo Hyun Ahn, Seungho Jeong, Jinsoo Lim, and Taegong Kim. Automated Transformation of Template-Based Web Applications into Single-Page Applications. In *IEEE 37th Annual Computer Software and Applications Conference (COMPSAC)*, pages 292–302. IEEE, 2013.

[103] Oracle, Java Platform, Enterprise Edition. http://docs.oracle.com/javaee/. Accessed: 2015-04-23.

[104] Terence John Parr. Enforcing strict model-view separation in template engines. In *Proceedings of the 13th international conference on World Wide Web*, pages 224–233. ACM, 2004.

[105] Linda Dailey Paulson. Building rich web applications with Ajax. *IEEE Computer Journal*, 38(10):14–17, 2005.

[106] Jesper Petersson. Designing and implementing an architecture for single-page applications in Javascript and HTML5. Master's thesis, Linköpings universitet, 2012.

[107] Chris Pilgrim. An investigation of usability issues in AJAX based web sites. In *Proceedings of the Fourteenth Australasian User Interface Conference (AUIC 13) - Volume 139*, pages 101–109. Australian Computer Society, Inc., 2013.

[108] Victoria Pimentel and Bradford G Nickerson. Communicating and displaying real-time data with WebSocket. *Internet Computing, IEEE*, 16(4):45–53, 2012.

[109] Play Framework Java Documentation, Session and Flash scopes. https://www.playframework.com/documentation/2.0/JavaSessionFlash. Accessed: 2015-05-06.

[110] Play Framework, Play 2.3.x documentation . https://www.playframework.com/documentation/2.3.x/Home. Accessed: 2015-05-14.

[111] React. https://facebook.github.io/react/. Accessed: 2015-04-24.

[112] ReactiveX, An API for asynchronous programming with observable streams. http://reactivex.io/. Accessed: 2015-05-07.

[113] Remy Sharp, What is a Polyfill? https://remysharp.com/2010/10/08/what-is-a-polyfill. Accessed: 2015-04-24.

[114] RequireJS: A JavaScript module loader. http://requirejs.org/. Accessed: 2015-06-06.

[115] RequireJS Documentation, Why web modules? http://requirejs.org/docs/why.html. Accessed: 2015-06-06.

[116] RxJava: Reactive Extensions for the JVM. https://github.com/ReactiveX/RxJava. Accessed: 2015-06-04.

[117] Chris Schaefer, Clarence Ho, and Rob Harrop. *Pro Spring*. Apress, 4th edition, 2014.

[118] Vibhu Saujanya Sharma, Shubhashis Sengupta, and KM Annervaz. ReLoC: A Resilient Loosely Coupled Application Architecture for State Management in the Cloud. In *2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, pages 906–913. IEEE, 2012.

[119] Joachim Haagen Skeie. *Ember.js in Action*. Manning Publications Co., 1 edition, 2014.

[120] Clinton W Smullen III and Stephanie A Smullen. An experimental study of ajax application performance. *Journal of Software*, 3(3):30–37, 2008.

[121] SockJS - WebSocket emulation. https://github.com/sockjs. Accessed: 2015-05-08.

[122] Steve Souders. High-performance web sites. *Communications of the ACM*, 51(12):36–41, 2008.

[123] Spring Engineering Blog, Spring MVC 3.2 Preview: Introducing Servlet 3, Async Support. https://spring.io/blog/2012/05/07/spring-mvc-3-2-preview-introducing-servlet-3-async-support. Accessed: 2015-05-11.

[124] Spring Framework Reference Documentation, View technologies. http://docs.spring.io/spring/docs/current/spring-framework-reference/html/view.html. Accessed: 2015-05-28.

[125] Spring Framework Reference Documentation, Web MVC framework. http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html. Accessed: 2015-05-11.

[126] Spring Issue Tracker, Support CompletableFuture as alternative to DeferredResult in MVC. https://jira.spring.io/browse/SPR-12597. Accessed: 2015-05-15.

[127] Michele Stecca, Luca Bazzucco, and Massimo Maresca. Sticky Session Support in Auto Scaling IaaS Systems. In *2011 IEEE World Congress on Services*, pages 232–239. IEEE, 2011.

[128] Luke Stevens and RJ Owen. The Truth About HTML5 Web Apps, Mobile, and What Comes Next. In *The Truth About HTML5*, pages 153–164. Springer, 2014.

[129] Adam Tacy, Robert Hanson, Jason Essington, and Anna Tökke. *GWT in Action*. Manning Publications, 2 edition, 2013.

[130] The Netflix Tech Blog, Reactive Programming at Netflix. http://techblog.netflix.com/2013/01/reactive-programming-at-netflix.html. Accessed: 2015-05-08.

[131] ThoughtWorks technology radar, JSF. http://www.thoughtworks.com/radar/languages-and-frameworks/jsf. Accessed: 2015-05-08.

[132] Stefan Tilkov and Steve Vinoski. Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010.

[133] Tim Berners-Lee, W3C. URI References: Fragment Identifiers on URIs. http://www.w3.org/DesignIssues/Fragment.html. Accessed: 2015-04-22.

[134] Twitter Engineering Blog, Implementing pushState for twitter.com. https://blog.twitter.com/2012/implementing-pushstate-for-twittercom. Accessed: 2015-05-06.

[135] Twitter Engineering Blog, Improving performance on twitter.com. https://blog.twitter.com/2012/improving-performance-on-twittercom. Accessed: 2015-05-06.

[136] Luis M Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45–52, 2011.

[137] W3C. Deep Linking in the World Wide Web . http://www.w3.org/2001/tag/doc/deeplinking.html. Accessed: 2015-04-21.

[138] W3C. Style Guide for online hypertext - Bookmarkable pages. http://www.w3.org/Provider/Style/Bookmarkable.html. Accessed: 2015-04-21.

[139] W3C. URIs, Addressability, and the use of HTTP GET and POST. http://www.w3.org/2001/tag/doc/whenToUseGet.html. Accessed: 2015-04-22.

[140] Qingling Wang, Qin Liu, Na Li, and Yan Liu. An Automatic Approach to Reengineering Common Website with AJAX. In *4th International Conference on Next Generation Web Services Practices*, pages 185–190. IEEE, 2008.

[141] WebJars - Web Libraries in Jars. http://www.webjars.org/. Accessed: 2015-06-07.

[142] Ming Ying and James Miller. Refactoring Traditional Forms into Ajax-enabled Forms. In *2011 18th Working Conference on Reverse Engineering (WCRE)*, pages 367–371. IEEE, 2011.

[143] Daniel F Zucker. What does AJAX mean for you? *Interactions*, 14(5):10–12, 2007.

List of Figures

List of Tables

List of Listings