Final Thesis

Rich Internet Applications for the Enterprise

A comparative study of WebWork and Java Web Start

by

Emil Jönsson

LITH-IDA-EX–07/063–SE

2007-12-07

Linköping University
Department of Computer and Information Science

Final Thesis

Rich Internet Applications for the Enterprise

A comparative study of WebWork and Java Web Start

by

Emil Jönsson

LITH-IDA-EX–07/063–SE

Supervisors: **Valérie Viale**
Amadeus

Philippe Larosa
Amadeus

Examiner: **Kristian Sandahl**
Department of Computer and Information Science
Linköping University

Abstract

Web applications initially became popular much thanks to low deployment costs and programming simplicity. However, as business requirements grow more complex, limitations in the web programming model might become evident. With the advent of techniques such as AJAX, the bar has been raised for what users have come to expect from web applications. To successfully implement a large-scale web application, software developers need to have knowledge of a big set of complementary technologies.

This thesis highlights some of the current problems with the web programming model and discusses how using desktop technologies can improve the user experience.

The foundation of the thesis is an implementation of a prototype of a central hotel property management system using web technologies. These technologies have then been compared to an alternative set of technologies, which were used for implementing a second prototype; a stand-alone desktop client distributed using Java Web Start.

Keywords: web development, Rich Internet Applications, WebWork, Java Web Start, Property Management System, hospitality software

Acknowledgements

First I would like to thank Amadeus for giving me the opportunity to do an internship at their development site in Sophia Antipolis. Secondly, I would like to thank the members of the inventory team at Amadeus and my unit managers Valérie Viale – for helping me to get started with the project – and Philippe Larosa – for helping me to finish it.

I would also like to thank professor Kristian Sandahl, my examiner at Linköping University, who has provided constructive feedback on the content and structure of the thesis. My opponent Daniel Johansson has also provided valuable feedback. I would also like to acknowledge my family for their encouragement.

Last, but certainly not least, I would like to thank the adorable Elizabeth Norris for helping me to improve the use of the English language in the thesis.

Contents

1 Introduction

1.1 Background

The idea for this thesis came up after having read an entry in Kurt Williams' blog – Cardsharp on Software. In the entry that caught my attention, Williams asked himself why "Rich Internet Applications"" have to be rendered in browsers, using a markup based approach.

> *Lately everyone in the industry is falling all over themselves to offer "Rich Internet Applications." We want our applications to be "richer." What does that mean? At its core, "rich" means "more like the GUI apps we used to write ten years ago. AJAX is on fire. Flex and Laszlo aren't far behind. We've even got SVG, Thinlets, and Nexaweb. The problem is that they all still use markup! That means you still have the mismatch. You still have to translate objects into text. And you still have a mountain of rendering layers to contend with.* (URL 23)

What Williams proposed is to use a technology for the Java platform for distributing software called Java Web Start. He then made an interesting comparison between the technology stacks needed for the traditional markup-based web applications compared to applications deployed using Java Web Start. Following are the technologies that Williams argued a developer needs to know to be able to create an AJAX enabled web application that displays data retrieved from a relational database:

> SQL/DB » Java » HTTP » XML » JavaScript/DOM » HTML » Browser

Compared to the above technology stack, these would be the technologies needed to create a corresponding application using Java Web Start:

> SQL/DB » Java » HTTP » Java

I found Williams' arguments very interesting and started to wonder why not more applications are written in Java and deployed using Java Web Start. Are we trying to reinvent the wheel? With the advent of the so-called Web 2.0 – web applications such as maps, spreadsheets and word processors have gained a lot of attention (Weiss, 2005), but do they actually bring something new? If so, could not the same features be implemented in an easier way using Java Web Start? Or, does Java Web Start bring so many new implications that it is not even worth considering?

The two approaches described above only takes into account what technologies are needed to display data retrieved from a database; what about other important issues such as security, internationalisation and usability? How are these issues addressed in the two different approaches?

This concoction of questions has inspired me to write this thesis.

1.2 Problem

From having been only about hypertext and exchanging documents, the web has changed into a medium allowing its users to retrieve and manipulate data in ways much like a traditional desktop application (Garrett, 2002). We are now using the web in ways it was not initially intended to be used in and sometimes our expectations change faster than the technologies themselves. Techniques such as AJAX have raised the bar for what users have come to expect from a web application. The gap between desktop and web applications is getting smaller.

Web application development often starts off simple, but as the business requirements become more complex, limitations in the web programming model might start to show. To successfully implement a large-scale web application, software developers need to have knowledge of a big set of complementary technologies and practices.

Stand-alone desktop application development offers a more advanced programming model, but potentially also more complex. How can we tell when the business requirements would be easier to implement using web technologies or when instead stand-alone client technologies would make a better fit?

1.3 Purpose

The purpose of this thesis was to study web application development in the context of a hotel front office application. On behalf of the travel technology company Amadeus, a working prototype of a central hotel property management system was implemented using web technologies. These technologies were then compared to an alternative set of technologies, which were used for implementing a second prototype; a stand-alone desktop client distributed using Java Web Start.

These are the questions this thesis is trying to answer:

- What problems can arise when developing web-based applications?
- How can these problems be addressed by instead developing a stand-alone desktop client distributed using Java Web Start?
- What problems and limits arise with the usage of Java Web Start?

1.4 Method

The study has been conducted as a case study, where the basis has been the experiences made during a six month internship at Amadeus in Sophia Antipolis in France. The technologies discussed are at the time of writing being used in real-world enterprise web application development. In a case study, realities are studied and explored to try to answer questions of the how- and why-type (Lindvall, 1997).

> *A case study is an empirical inquiry that investigates a contemporary phenomenon within its real-life context; especially when the boundaries between phenomenon and context are not clearly evident.* (Yin, 2003)

While working on a project for Amadeus to develop a prototype web application, problems that occurred during the implementation phase were documented. These problems have then been analysed in order to see how they could be addressed using the Java Web Start technology. To make a fair comparison, current problems existing with the Java Web Start technology and how these problems could be addressed using a browser-based approach have also been studied.

To find relevant printed literature, online article databases accessible through the Linköping University library have been used. It is important to remember the fact that these kinds of technologies are constantly changing, making it harder to find relevant and up-to-date literature. Some of the information used for this thesis has been found on the internet on documentation web sites, in blogs, user forums and sometimes even in source code repositories.

At the time of writing only one printed book solely dedicated to WebWork existed and for Java Web Start the situation was almost the same. A search on Amazon.com using the keywords "java web start" only resulted in two books, published in 2001 and in 2002, respectively.

1.5 Limitations

There is not only *one* way to create a browser-based web application, a rich flora of programming languages and frameworks exists and it is futile to argue over which one of these frameworks is better than the other. The Microsoft .NET platform, LAMP (Linux, Apache, MySQL and PHP) and Ruby on Rails are all well-known examples of web technologies but, to limit the scope of the thesis, focus has been kept on the Java platform, even though some of the web development problems discussed applies to web applications in general. Only for Java, a multitude of web frameworks exists (Kabanov & Mürk, 2006), why the scope has been even further limited to compare solely the WebWork framework with the Java Web Start technology. By comparing two Java-based technologies we also avoid the discussion of which programming language is best suitable for Rich Internet Application development.

1.6 Structure

The structure of this report has been based on the recommendations in *Lathund för rapportskrivning* by Andersson et al (2006).

The introductory chapter explains the background and motivation for the thesis. It also describes the method used for the study and the trade-offs that have been made. In the second chapter, the reader is given a more detailed background of the technologies and concepts discussed in the thesis.

The subsequent chapters describe how the two prototypes were developed, along with a comparison of the two approaches. Finally, the last two chapters contain personal reflections and a conclusion.

The acronyms used throughout the thesis are listed along with their full names on page 47. There is also a glossary available on page 45.

2 Preliminaries

2.1 Web technologies

A web application is typically created using several complementary technologies and this chapter describes the basic building blocks.

2.1.1 HTML, CSS and JavaScript

The web was initially created as a way for researchers around the world to share information and its core was hypertext documents (Garrett, 2002). Documents written in the *Hypertext Markup Language* (HTML) could be published online and also contain links to other documents. With time, the web changed from being a static collection of information into the dynamic medium we are now used to. HTML documents are still the core of the web, but today it is also possible to embed images, videos and other types of interactive content in a web page.

Cascading Style Sheets (CSS) is a formatting language that has been created to enable developers to control the visual presentation of web documents. A style sheet typically contains several rules, each of which consist of a selector and a declaration block. The selector tells to which elements the rule is to be applied, while the declaration block contains a set of CSS properties with values. A basic example of a property is the colour property, which can be used to override the browser's default text colour. By using the same external style sheet for all documents, the look of an entire website can be changed by editing just a single source file. (Meyer, 2004)

JavaScript is an object-oriented programming language which can be used to enhance the functionality of a web page. For example, client-side validation of user input can be performed using JavaScript and as one of the cornerstones of AJAX (see 2.1.3), the language has recently come to gain more popularity. The implementations of JavaScript vary across different browsers and this is something that has caused developers a lot of problems. However, by using one of the many JavaScript libraries available, developers can more easily overcome the issues related to cross-browser compatibility. (Almaer et al., 2006)

2.1.2 Web standards

The World Wide Web Consortium (W3C) was founded in 1994 by Tim Berners-Lee, with the goal of developing common protocols to ensure the interoperability of the web (Cederholm, 2004). In the beginning, the specifications produced by W3C were not always followed by browser manufacturers. The companies behind the two most popular browsers –

Microsoft and Netscape – kept adding their own extensions, which lead to the so-called Browser Wars (Holzschlag & Shea, 2005).

Over time, the major browser manufacturers started to standardise on the W3C specifications. This made it possible for web developers to create web pages that would look reasonably consistent no matter what browser the user was using (Holzschlag & Shea, 2005). It was no longer necessary to use deeply nested table structures or spacer GIF shims to achieve the desired document layout (Cederholm, 2006).

To adhere to web standards means following the specifications from the W3C, more specifically CSS, HTML 4.01, XHTML and the Document Object Model (DOM) (Holzschlag & Shea, 2005). Of the latter two, XHTML is an evolution over HTML which adds a few more rules to make it XML compliant, while the DOM is an interface for accessing and manipulating the content, structure and style of documents (Zeldman, 2003).

"No presentational markup" has become the mantra of standards aware web developers. By using only structural markup, the content and the presentation of the web page get more clearly separated. Updating and redesigning become easier when using CSS to control the presentation and as earlier mentioned a single style sheet can control the look of an entire website. Also, the specifications produced by W3C have been written with future compatibility in mind and to make the content available to a wide range of user agents. (Cederholm, 2004)

To summarise, these are four benefits of using web standards as identified by Cederholm (2004):

- Reduced markup
- Increased separation of content and presentation
- Improved accessibility
- Forward compatibility

The importance of using web standards has grown since today we are accessing the internet not only from desktop computers, but also from handheld computers and mobile phones. By adhering to web standards a website can degrade nicely when the user's browser only has a limited set of features. It might not look exactly as initially intended, but the content will still be available. (Zeldman, 2003)

2.1.3 AJAX

Just like with web standards, neither AJAX is a technology in itself, but rather a way to combine already existing technologies. The acronym AJAX is short for *Asynchronous JavaScript and XML* and the term was coined by Jesse James Garret – usability expert at Adaptive Path – in 2004 (Keith, 2007). Before the term had been coined the same set of technologies was referred to as *remote scripting* or *remote JavaScript* (Schwerin, 2006).

An AJAX application is in its most basic form an HTML user interface where a JavaScript function gets triggered, for example by a timer or an action made by the user. Using the browser's built-in XML HTTP request object, the JavaScript function sends an asynchronous request to a web server and sets a call-back function. The server then generates an XML response and sends it back to the browser. When the browser retrieves the response it calls the JavaScript call-back function, which was specified when the request was initiated. The call-back function can then update the web page with the data returned from the server by manipulating the DOM. (ibid)

In practice, neither the browser's request nor the response from the server have to necessarily be encoded in XML. The XML HTTP request object can rather be regarded as a wrapper for making HTTP requests. For example the server can return HTML fragments or any other arbitrary data format. (Heinemeier Hansson & Thomas, 2005)

By making use of AJAX the need for full-page reloads can be reduced, providing a way to make partial updates of a web page. As a result the web page gets more responsive, which in turn can improve the user experience. At the same time AJAX has also come to be synonymous with richer HTML user interface elements, such as sliders and accordion panels. (Schwerin, 2006)

2.2 Java Web Start

Java Web Start (JWS) is a platform-independent deployment technology for launching Java applications distributed over the web (URL 19). Updates of an application are downloaded transparently, making sure that the user always has the most recent version of the application. To be able to launch a JWS application, the user needs to have an appropriate version of the Java Runtime Environment (JRE) installed. JWS itself is included in the JRE distribution.

In order to deploy an application using JWS, a Java Network Language Protocol (JNLP) file needs to be created (Bates & Sierra, 2005). Basically, the JNLP file holds metadata about the application and its dependencies, such as which main class to invoke to run the application and whether offline mode is allowed or not (see appendix C, page 59, for a sample file). A standard web page with a link to the JNLP file can then be created and provided that the web server is aware of the JNLP MIME-type, the user can launch the application by clicking on the link (ibid). JWS will then download the application and run it outside the browser. For subsequently running the application, JWS can automatically create shortcuts (URL 20). The user can then launch the application without needing to open a web browser. Another option is to use the JWS Cache Viewer for launching applications (ibid).

When a JWS application is launched, it runs in a sandbox on the user's computer. For security reasons, applications running in the sandbox have restricted access to local resources. If an application needs access to the file system or the network, it can be digitally signed with a certificate. The first time the application is launched, the user will be presented with a dialog box asking the user to accept the certificate. If accepted, the application will be allowed to run outside of the sandbox. (URL 19)

The need for an appropriate JRE installed on the user's machine is one major drawback of JWS (Marinilli, 2006). In an article on Sun's website, Haase et al (URL 5) discuss how to launch a Java application from the browser, along with possible issues. They also propose a solution using client-side scripts to detect whether JWS is installed or not and to start an installation if necessary.

2.3 WebWork

WebWork is an open source web application framework with built-in user interface components and support for type conversion, form validation and localisation.

2.3.1 Background

The first publicly available version of WebWork was released in the fall of year 2000 and had been developed by a single developer. As the framework gained more momentum it later joined the OpenSymphony group and became a community effort. When initially created, the goal of WebWork was to build a framework where the correct way to solve a problem should also be the easiest. This is still a core philosophy of WebWork. (Carreira & Lightbody, 2005)

2.3.2 An action-based MVC framework

When using a framework, a developer is forced to work within certain boundaries, adhering to the structure provided by the framework. This in turn increases the maintainability of the application, assuming the framework helps the developer to divide the application into separate parts with different responsibilities and loose coupling between them.

Two such parts with different responsibilities are the domain model and the presentation. The issue of decoupling these two parts is not specific to web programming and many web frameworks are using a pattern which has evolved from the desktop programming world called the Model-View-Controller (MVC) pattern.

In the MVC pattern, a controller receives events, such as form submissions, from the view and then performs actions on the model. It is also the responsibility of the controller to notify the view when changes occur in the model so that the view can be updated accordingly. The pattern decreases the coupling between the view and the model and makes it possible to modify the view without having to modify the underlying model.

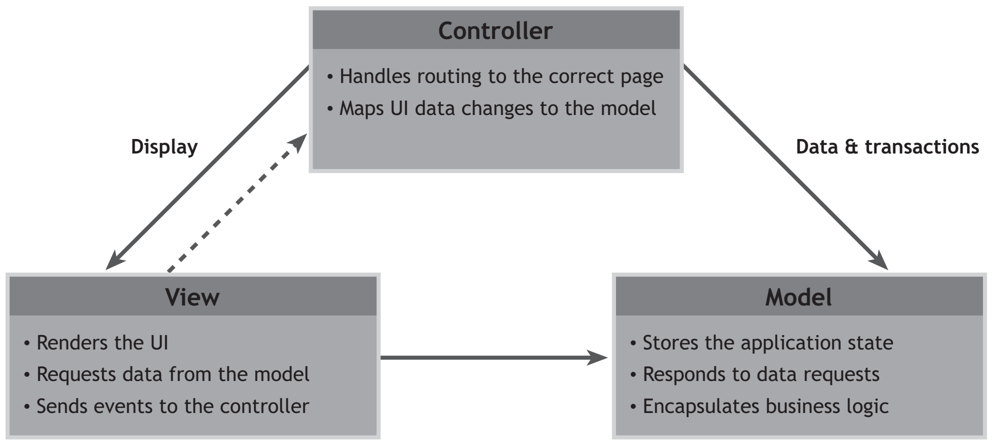
Controller
• Handles routing to the correct page
• Maps UI data changes to the model
Display
Data & transactions
View
• Renders the UI
• Requests data from the model
• Sends events to the controller
Model
• Stores the application state
• Responds to data requests
• Encapsulates business logic

Figure 1 – The MVC pattern as applied to web frameworks (Mills, 2003)

For a web application, the view resides client-side in the user's browser, while the controller and the model reside server-side. If the model changes, it is not possible to propagate these changes to the view due to the static nature of the web[1]. For the changes to be reflected in the view, a new request has to be made from the user's browser.

The fact that the view resides client-side implies that direct calls to the controller can not be made. Instead, a view is mapped to a controller based on a request URL. This URL is then handled by a so-called dispatcher, which is responsible for mapping the URL to a command. This is the traditional

[1] This is not strictly true. Though not commonly adopted, there is a technique called AJAX Push or Comet where long-lived HTTP connections are used to allow for the server to send data to the client on demand.

way of applying MVC for web frameworks and it is known as the *front controller* pattern. The dispatcher can be seen as a master controller responsible for routing requests to sub-controllers (commands). For WebWork applications, the dispatcher is implemented as a Java servlet and the commands executed by the dispatcher are called *actions*.

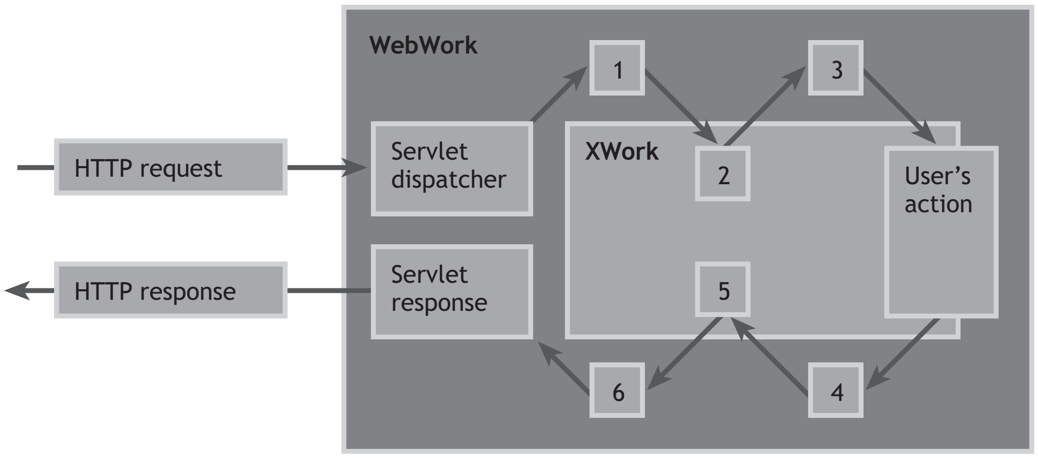

Figure 2 – The flow in a WebWork application (Carreira & Lightbody, 2005)

Figure 2 shows how an HTTP request issued from the user's browser is handled by the WebWork servlet dispatcher and routed through a set of *interceptors* (see 2.3.4). Then the action is executed and the result is routed through another set of interceptors. Finally, via WebWork's servlet response, the result is converted to an HTTP response and returned to the user.

2.3.3 Architecture of a WebWork application

For the interested reader who finds it easier to understand a concept by looking at source code rather than reading about it, the source code of a sample WebWork application can be found in appendix B on page 55.

As mentioned earlier, WebWork's servlet dispatcher is responsible for routing HTTP requests to actions. The mappings between the requests and the actions are configured in a file called `xwork.xml`. In the same file it is also configured what interceptors to apply before the action is executed and how to handle the action's result.

Actions are implemented as Java classes and contain the logic needed to handle requests made by users in a meaningful way. They act as interfaces for views to the application's Java code base. For example, in an application where the user can edit the content of a database through an HTML form, the action's properties will be set with the user's input when the user submits the form. It would then be the responsibility of the action that these changes are somehow propagated to the database. At the same time it can also work the other way around, where view templates retrieve data to be displayed from the actions.

When the action has finished executing it returns a result string, which is then used by the servlet dispatcher to determine what kind of result to return to the user. Most commonly, a JSP page will be rendered and an HTML page will be returned to the user. Other options include returning a PDF document or an XML file. Developers can also define their own results.

Apart from reading data made available through an action, WebWork can be configured so that views can read data from Java property files. This feature is useful when creating a localised application. Translations of the various texts used throughout the application can be stored in different property files and retrieved when needed.

2.3.4 Interceptors

The concept of interceptors is a feature of WebWork that makes it different from other Java-based web frameworks. Interceptors allow the developer to add code to be processed before and after an action is executed as a way to modularise common functionality. Much of the core functionality of WebWork is implemented using interceptors and it is also possible for developers to add their own interceptors.

To gain an understanding of what interceptors can be used for, a basic example could be a timer interceptor. The purpose of the timer interceptor would be to track how long time it takes to execute an action. When the action for which the execution time is to be tracked is called, the timer interceptor gets executed before the action and saves a timestamp. After the action has been executed, control returns to the timer interceptor and by using the timestamp the action's execution time can be calculated.

Another example is the *params interceptor*, which is one of the interceptors built-into the framework. When used, the parameters from the request are used to set the corresponding properties of the action. The parameters are automatically converted to the type of their respective property. This is how data submitted from an HTML form is made available to an action.

WebWork includes interceptors for logging, validating form data, preparing resources needed by the action and more. To make it easier for developers to add their own interceptors there are also interfaces provided. Developers can implement one of these interfaces, add the code specific to the interceptor and then use it in an application.

The order in which interceptors are executed is specified in the configuration settings and it is possible to group several interceptors into an *interceptor stack*. The WebWork distribution comes with a set of predefined interceptor stacks, but it is common to extend these to fit the specific needs of the application.

2.3.5 Form validation

When creating an application it is important to design it in a way that prevents the user from making mistakes (Plaisant & Shneiderman, 2005). In a form-based application where the user is supposed to enter data, it is desirable to catch errors made by the user as early as possible. This is why form validation is an important feature of a web framework. Validation can for example be used to make sure an entered integer value is within a given range or that an entered value matches a certain pattern.

WebWork provides the developer with several ways of performing form validation. Maybe the simplest way is to add the validation logic directly in the action's *execute()* method. The main drawback of this approach is that it limits the possibilities of reusing the validation logic. However, the approach can still be useful in cases where the business rules are complex and the alternative approaches listed below do not fit.

Another approach is to have the action class implement the *Validatable* interface. This interface requires the developer to implement a *validate()* method, which is where the validation logic is to be put. The validation logic can then be refactored out of the *execute()* method and into the *validate()* method, making the separation of business logic and validation logic clearer.

The final approach to form validation is to use the built-in validation framework. Rather than expressing the validation rules as code, the framework allows for the validation logic to be externalised and represented as validation metadata in XML. The standard distribution of WebWork comes bundled with a set of predefined validators. These validators can be used to validate for example required fields, dates and URLs. Developers can also add their own validators to be used with the validation framework.

2.3.6 Testing

To create a framework where the application code was easily testable was one of the goals when WebWork was created. As a core principle of WebWork, this has led to a design where the action classes have no dependencies on the Servlet Application Programming Interface (API). The benefit of this loose coupling is that action classes can be tested as plain Java objects without having to be run in a servlet container.

In a unit test, an instance of the action class can be created and the property values of the action set as needed. The test can then execute the action and verify that it behaves as expected, for example that the correct result string is returned.

As the actions grow more complex and start having external dependencies, a strategy can be to use so-called *mock objects*[2]. The idea of mock objects is to create simpler versions of the objects on which the class to be tested is dependent, allowing for the class to be tested in isolation.

2.3.7 The future of WebWork

As of writing, WebWork is being merged with the widely adopted Struts framework. The project is hosted by the Apache Software Foundation and the outcome of the merge will be Struts 2, which to a large degree is based on the WebWork code base. (URL 4).

2.3.8 Alternative web frameworks

Java is a common platform for server-side web development and in 2006 more than 30 open source web frameworks for Java existed (Kabanov & Mürk, 2006). For a comparison of some of the more popular ones, see Næssén (2007).

2.4 Rich Internet Applications

Traditional HTML-based web applications got popular much thanks to low deployment costs. Other factors were the simple architecture and the fact that HTML in itself was easy to learn. The simplicity offered by HTML-based interfaces made it worth to sacrifice some of the application responsiveness and user experience. (O'Rourke, 2004)

Later, technologies have emerged to address the shortcomings of HTML-based web applications and to provide richer user interfaces. Examples of these are Adobe Flash and client technologies for both the Microsoft and the Java platforms (O'Rourke, 2004; Marinilli, 2006). Collectively, the applications created using these technologies are called Rich Internet Applications (RIA). Commonly, AJAX is also included as a technique for developing RIAs (Handy, 2005; Loosley, 2006; Schwerin, 2006).

Rich Internet Applications are a cross between web applications and traditional desktop applications, where some of the processing takes part client-side while the rest of the processing is kept on the application server

[2] See for example EasyMock (www.easymock.org) or jMock (www.jmock.org)

(Loosley, 2006). RIA technologies allow for applications to be deployed over the internet with the same simplicity as for web applications (O'Rourke, 2004).

The above definition of a RIA is rather brief, what does it actually mean for an application to be rich? In a Macromedia white paper, Mullet (2003) mentions four qualities, identified by the Macromedia Experience Design team, that distinguish rich interactive experiences.

Seamless
Focused
Connected
Aware

Figure 3 – Four qualities that distinguish rich experiences (Mullet, 2003)

A *seamless* experience means that the application provides immediate responses and a continuous workflow. The interaction in traditional web applications is typically not seamless as full-page reloads are often required, forcing the user to wait for the next page to be rendered. (Mullet, 2003)

When an application targets a single task, the experience can be said to be *focused*. The application should not display any unnecessary information and also prevent users from making errors. (ibid)

Applications designed to be *connected* offer their users transparent access to the resources they need, saving them from the hassle of managing the connections themselves (ibid). Laptop computers are becoming more and more popular and by 2005, more laptops than desktop PCs were sold (Weiss, 2005). Combined with the increase in wireless networking, this makes the physical location of the user less important (Mullet, 2003).

The final identified quality is *awareness*, which means that the application seems to be aware of the task the user is trying to achieve. For example, data entered during previous usage of the application can be reused when the application is subsequently launched to save the user from repetitive typing. Another example could be an application in which the user interface language is adapted to the user's current locale settings. (ibid)

All of the listed qualities are not necessarily needed for every application and the development budget might not allow for all of them to be realised. However, having a user interface that gives the user a richer experience can be a competitive advantage. (ibid)

2.5 View technologies

Several technologies exist for enhancing the user experience and developing richer applications, in this thesis WebWork and Java Web Start have been studied. The related technologies listed in this section are interesting alternatives, but further investigation is however outside the scope of the thesis.

2.5.1 Flex

The Flex Application Framework has been developed by Adobe and is primarily concerned around the application's presentation layer (Brown, 2007). Adobe's main goal with Flex is to provide a rapid user interface development platform and Brown (2007) claims that Flex is the next evolutionary step for Flash. The Flex Standard Developer Kit (SDK) is available free of charge, but Adobe also offer commercial products such as additional components and an Eclipse-based IDE, Flex Builder (URL 1).

The foundations of the Flex Application Framework are MXML and ActionScript 3.0. MXML is an XML-based markup language used to arrange the visual elements and to provide structure for the application. ActionScript has evolved from being a limited programming language, used mostly to control the playback of Flash movies, into an object-oriented programming language with a syntax similar to Java's. In Flex, ActionScript is used to add interactive behaviour to the components. Conceptually, the way MXML and ActionScript are combined is analogous to how HTML and JavaScript are used together. Just like for the latter combination, the look of the user interface can be controlled using CSS. (Brown, 2007)

In order to have the Flex layer communicating with the application's backend layer, built-in components called *Flex Data Services* (FDS) can be utilised. Using these components Flex can connect to web services and either retrieve content to be displayed or send data which is to be processed server-side. This means that a user interface implemented in Flex can leverage services implemented in C#, Java, PHP or any other dynamic server-side language. The FDS components also support an offline mode, allowing the user to temporarily work with data without being connected to the internet. (ibid)

When a Flex application is to be deployed, the MXML is compiled to a format that can be understood by the Flash Player. Provided that the user has the proper version of the Flash Player installed, the application can be viewed in a standard web browser (ibid). Adobe is also working on a cross-platform runtime environment called Air for deployment to the desktop, which is currently available for download as a beta release.

The goal of Air is to let developers use their existing skills in web development – Flash, Flex, HTML, AJAX – to build and deploy desktop applications. The strengths of web applications, such as network connectivity, rich media content and ease of deployment, can be combined with the benefits of desktop applications, such as access to local resources and interaction with other applications. (URL 2)

2.5.2 OpenLaszlo

OpenLaszlo is an open source view technology originally developed by Laszlo Systems and the same company is also the main sponsor of the project. Similar to Flex, an XML-based language is used for arranging the visual elements of the application. The language, called LZX, also includes a subset implementation of JavaScript which can be used to add interactive behaviour to the interface. (URL 16)

An API for using remote services, OpenLaszlo RPC, is included in the distribution and there are currently three RPC implementations available: Java RPC, XML-RPC and SOAP. The API allows for OpenLaszlo applications to communicate with various kinds of back-end services. (ibid)

Two different output formats are supported when deploying OpenLaszlo applications. Consequently, the same code base can be used to produce an application in either Flash or DHTML format. The application can then be run in a standard web browser. Laszlo Systems is also working together with Sun Microsystems on a project called Orbit, which aims to enable OpenLaszlo applications to run on mobile devices equipped with Java Micro Edition. (ibid)

2.5.3 Silverlight

Silverlight is a view technology developed by Microsoft. The aim is to enhance the web programming model by letting developers use a subset of the Windows Presentation Foundation (WPF). The Silverlight runtime environment is available free of charge and Microsoft also sells proprietary products for developers who are using the technology. (URL 11)

Applications are written in an XML-based language, eXtensible Application Markup Language (XAML), which is the same view markup language as in the Microsoft .NET Framework 3.0. Silverlight is not tied to any specific back-end technology, allowing for integration with existing applications and infrastructure. On the client-side, it is possible to interact with the Silverlight application using JavaScript, since the runtime environment exposes the XAML document to the browser. (ibid)

As of writing, version 1.0 of the Silverlight browser plugin is available for download and currently three browsers are supported – Firefox and Safari on both Mac and Windows, and Internet Explorer on Windows. (ibid)

2.5.4 XUL – XML User Interface Language

The XML User Interface Language (XUL) comes from the Mozilla Project. It has been developed with platform independence in mind to make applications easily portable to the various platforms on which Mozilla applications run. For example, the user interfaces for the Firefox browser and the e-mail client Thunderbird are written in XUL. (URL 12)

The language is based on W3C's XML 1.0 specification and XUL applications typically make use of several other standard technologies, such as HTML, CSS and DOM. XUL provides the developer with widgets such as menus, tab panels and hierarchical trees. Like for a general HTML application, the presentation and behaviour of a XUL application can be controlled using CSS and JavaScript respectively. XUL has support for localisation and built-in APIs are available for accessing remote content or web services. (ibid)

A XUL application can be run directly in one of the Mozilla browsers. Another option is to bundle Mozilla's Gecko rendering engine with the application, making it possible to deploy standalone desktop applications. (URL 23)

Currently, work is ongoing to create a Gecko runtime environment to allow for XUL applications to be deployed without bundling the Gecko engine. The goal is to have the runtime environment able to be downloaded and installed automatically as the application is being installed, unless the runtime is already available on the user's system. (ibid)

3 The internship project

This chapter presents the background of the internship project on which this thesis has been based and how the project was carried out.

3.1 Amadeus

Amadeus is a world-leading technology company in providing IT solutions for the travel and tourism industry. It was founded by Air France, Lufthansa, Iberia and SAS in 1987 to create a Global Distribution System. (URL 3)

The company works with travel providers such as airlines, hotels and tour operators, but also directly with travel agencies. Currently, more than 89.000 travel agency locations and 29.000 airline sales offices are using the product Amadeus System for running their business. (ibid)

The corporate headquarter is located in Madrid in Spain and the development site is located in Sophia Antipolis in France. Amadeus also has one of the world's largest data processing centres dedicated to travel, located in Erding in Germany. (ibid)

3.2 Background

The internship was carried out in the inventory team at the Hotel Central Reservation Services development department in Sophia Antipolis. The department is responsible for the development of the Amadeus Hotel Central Reservation System (CRS) and the object of the internship was to investigate future phases of this project.

More specifically, the goal of the internship was to develop a prototype of a central property management system (PMS) for hotels. A hotel PMS is a front office application used directly by the hotel staff to provide effective service for the guests. Due to the extent of a hotel PMS, the internship was focused on two specific use cases: *guest check-in* and *guest check-out*.

The project was divided into three phases, starting with a specification phase where software requirements were gathered and documented in a product specification document. Then, in the design phase, a high level design document where the various requirements were taken into account was produced. The final phase was the implementation phase, where a working prototype of the hotel PMS was realised.

A timetable for the internship project can be found in appendix D on page 61.

3.3 Specification

As the first phase of the project, the specification phase started out with information gathering about the hotel business in general and also about existing PMS solutions on the market. This was done to gain a better understanding of what a PMS would be used for and to help define the requirements of the prototype. Also, the projects within the inventory team were presented.

The two main use cases, *guest check-in* and *guest check-out*, were broken down into several more specific use cases. Depending on its function, each use case was grouped as either a user interface use case or a business domain use case. The use cases were then modelled as use case diagrams in UML using Rational Rose to show how they were related (see appendix A, page 53).

A known problem – common in many of the current PMS solutions – is how customer profiles are handled. For the more exclusive hotel chains, it can be of interest to keep track of specific customer preferences in order for the staff to be able to provide a better service. When a new booking is made, it is possible that a profile of the customer already exists, but the staff responsible for making the booking might not find this particular profile in the system. Instead, a new customer profile is created and the system now contains duplicates. Potentially valuable information about the customer known from previous stays might not be available for the next stay. To address this problem, some hotel chains have employees who are responsible for finding and merging customer profiles in those cases where several profiles have been mistakenly created for the one and same customer. Although it was not the object of the internship to completely solve this issue, it was still taken into account. To help minimise the possibility of creating duplicate customer profiles, some specific profile handling features were introduced in the prototype.

Another interesting fact that came up during the requirement study was that having a high staff turnover rate is common within the hotel business. This resulted in a requirement to design the prototype with user interface plasticity in mind, to make it have a low entry level for beginners while providing alternative ways of using the software for more experienced users (Plaisant & Shneiderman, 2005).

The specification phase resulted in a product specification document which then had to be approved by the department manager before the next phase could be started.

3.4 Design

There was a requirement from Amadeus to use an architecture similar to the one being used within the Hotel CRS project. Advantage could then be taken of toolkits and libraries that had already been developed by the project teams. This meant that the prototype was to be designed as a back-end application with a web-based front-end. To realise the communication between the front-end and back-end servers, the application would leverage a transaction framework developed in-house by Amadeus.

The nature of a hotel PMS, where users often have to enter customer information, made it evident that form-filling would be the main interaction style of the application (Plaisant & Shneiderman, 2005). Together with the requirement of having an application with user interface plasticity, this lead to the decision that a driving goal would be to design the user interface in such a way that the user would have to use the mouse as little as possible. Experienced users would then be able to navigate the application using the keyboard as the primary interaction device, while still allowing for novice users to use the mouse to interact with the application.

At this stage, a high level design document was written to detail how the system would work and how the requirements were to be implemented. The design document also contained UML interaction diagrams showing the control flow between the various objects and actors in the system.

3.5 Implementation

The third phase was the actual implementation, which is described in the subsequent chapter.

4 The first prototype

4.1 General architecture

The prototype application was implemented following a service-oriented architecture (SOA) approach with a graphical user interface (GUI) developed as a Java web application and a back-end layer with stateless services. The communication between the user interface and the back-end was realised by sending OTA[3] XML messages wrapped in EDIFACT[4] messages using a transaction framework with additional libraries developed by Amadeus.

4.2 The user interface

The user interface was implemented as a browser-based application in Java using the WebWork framework. This meant that the prototype could be run in a standard web browser, such as Mozilla Firefox or Internet Explorer.

To fulfil the requirement of user interface plasticity, the user interface was built in such a way that three operations which were likely to be the most commonly used could all be performed from the prototype's main screen. These operations were *guest check-in*, *guest check-out* and *make new reservation*. Experienced users would then be able to perform common operations without having to switch to another screen. New users, on the other hand, would still be able to switch to a screen specific to the operation they wanted to perform.

Two AJAX features – auto-completion and data filtering – were implemented in the GUI using the MochiKit JavaScript library (URL 10). These features were both introduced as a partial solution to the aforementioned problem with customer profile duplication. As a new reservation was being made, the receptionist could see a list of available profiles matching the customer data just entered. If the receptionist found that one of the listed profiles belonged to the customer for who the reservation was being made, this profile could be selected. The empty fields in the reservation form were then automatically filled out with data from the selected profile.

[3] OpenTravel Alliance (OTA) is a non-profit organisation working on creating open e-business specifications for the travel industry (URL 17).

[4] EDIFACT is a data-interchange format for administration, commerce and transport (URL 21)

4.3 The service back-end

The back-end layer of the application was implemented as a set of stateless C++ services. One of the major benefits of this architecture is scalability. If a particular service is being used extensively, more instances of that service can be started to increase the performance. Routing of service requests to a corresponding service instance is handled transparently by the transaction framework. If each service is designed with a well-defined purpose, the service-based approach can also lead to a system with loosely coupled components, something which is often desirable as it makes maintenance and reuse easier.

The services in the PMS prototype were designed to handle database communication. The Java front-end never used the database directly, but instead retrieved data by communicating with the back-end services. For each of the application's domain objects, for example *Reservation* and *Profile*, two services were implemented. The first service was used to either retrieve a specific object based on a unique identifier or for retrieving all objects of that specific type. The second service was used for creating, updating and deleting objects. As an example, this means that for *Profile* domain objects, there was one *RetrieveProfiles* service and one *ManageProfiles* service.

It should be mentioned that a back-end service performing create, read, update and delete (CRUD) operations on data stored in a database is probably one of the simplest scenarios for a service. Services can be designed to perform considerably more complex tasks and can for example leverage other services.

4.4 The communication flow

The figure below shows the communication flow in the prototype when an event in the user interface has triggered an AJAX call. Using the browser's built in XML HTTP request object, a request is sent to the web server acting as the application's front-end server. Based on the URL, the request is mapped to a WebWork action, causing the action to be invoked.

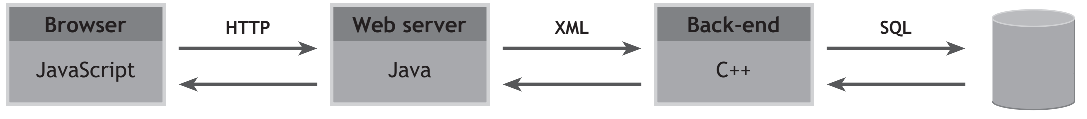

Figure 4 - The communication flow in the prototype application

In the WebWork action object, an XML message for querying the back-end is constructed. An XML processing library, XMLBeans, was used to create Java class representations of the XML schemas that defined the OTA XML messages to be sent between the front-end and the back-end servers. These Java classes were then used to construct the XML messages.

The XML message is then wrapped in a generic EDIFACT message before it is sent to the service back-end. The reason for this is that the framework that handled the communication between the front-end and the back-end used EDIFACT as the data interchange format. In the back-end the message is routed to the corresponding service. The service retrieves the requested data from the database and sends a reply back to the WebWork action. From there, an HTTP response is generated and sent back to the user's browser.

4.5 Problems with the implementation

During the implementation of the prototype, the three problems listed below were revealed. In chapter six (6.1), the problems and their implications are described in detail.

- JSON versus HTML fragments
- Field gets validated when a type error has occurred
- JavaScript Hijacking

5 The second prototype

The object of the thesis was to compare two ways of distributing software over the web, first by implementing the prototype as a web application and then to create a Java Web Start enabled desktop application.

5.1 *Scope*

Due to limited time it was not possible to fully implement a second version of the prototype. Instead, only a few of the use cases were implemented, meaning the second version of the prototype is to be seen more as a proof-of-concept implementation.

As was mentioned in the introduction chapter, the idea was to address problems with the web-based prototype when creating the second implementation. However, the problems that occurred during the original implementation were not directly related to the limits of the web-based technologies. What took time was to correctly understand the problem domain and to learn the libraries and toolkits that were to be used.

One of the more advanced features of the web-based interface was the start screen, from which the user can create new reservations, check-in and check-out hotel guests. When the user starts entering data in the screen's form, the lists of expected check-ins and check-outs are automatically filtered and entries that do not match the data just entered are removed. In the original prototype this feature was implemented using AJAX and it was decided that the same feature was to be implemented in the second version of the prototype.

5.2 *Implementation and deployment*

Some of the code from the original prototype could be reused, more specifically the model objects and the data access layer. The development of the second implementation took place after the internship at Amadeus had finished and for this reason the transaction framework and the service-based back-end was no longer available. However, thanks to having used the Abstract Data Access Object (DAO) Factory Pattern to encapsulate all the data storage access (Alur et al., 2001), this was never a problem. Instead, the service-based DAO factory was swapped with an in-memory DAO factory that was developed during the implementation of the web-based front-end.

The user interface was implemented as a basic form-based interface using Swing, Java's built-in GUI toolkit. When the application was working as expected, the next step was to deploy it.

The following are the steps that were needed to deploy the application with Java Web Start:

- Create an executable JAR file
- Create a JNLP configuration file
- Publish the two files on a web server
- Configure the web server to send JNLP files with the mime type "application/x-java-jnlp-file"
- Create a web page with a link to the JNLP file
- Done. Users with Java Web Start installed can now launch the application directly from the web page.

The JNLP configuration file is a central part of a JWS distributed application and contains meta-data for specifying application and distribution properties (see appendix C, page 59). On the computer on which the tests were to be performed, the Internet Information Services (IIS) web server from Microsoft was available and the application was configured to be started via a locally installed server. To publish the application files, a new virtual directory pointing at the directory where the JNLP file and the JAR file were located was created in IIS.

5.3 Problems

The first attempt to start the application failed. When the URL to the JNLP file was entered in the browser, the file was downloaded and Java Web Start was launched. A download window was displayed, showing the JAR file download progress. Since the tests were performed using a locally installed web server and the file was small, the download took literally no time at all. However, when the JAR file had been downloaded and JWS tried to execute the actual application, it failed with an error message telling that the main class could not be found. The problem was easily fixed; it appeared there was an error in the JNLP configuration file. Instead of specifying the full path of the main class, only the class name had been stated.

After having changed the erroneous parameter in the JNLP file, a new attempt to start the application was made. Yet again, JWS was launched but failed to execute the application. The application's splash screen was shown for a few seconds and then nothing else happened. This time there was no error message giving a hint of what could have gone wrong.

The problem appeared to be a missing library file for a runtime dependency; the code in the concrete DAO implementations needed a logging library JAR file. By adding a resource parameter in the JNLP file and making the logging library available on the web server the problem was solved.

5.4 The automatic update feature

A basic test was performed to test the automatic update feature of Java Web Start. After having successfully launched the application, the web server was turned off to make sure that the application files could not be reached via the network. Some minor changes were then made to the application to make it easy to verify what version of the application which was running.

With the web server still turned off, the application was launched again and since it had been configured to allow for offline usage, it was started using the cached versions of the required application resources.

After it had been confirmed that the application worked offline, the web server was started again. When launching the application this time, Java Web Start could reach the server where the application resources had been published. The latest version of the application JAR file was downloaded and then the application was started.

6 Evaluation

In this chapter, the problems that arose during the development of the two prototypes are further analysed. For the sake of simplicity, the prototype implemented using WebWork will hereafter be referred to as P1 and the standalone Java client prototype as P2.

6.1 Problems with the initial prototype

6.1.1 HTML fragments versus JSON

The problem whether to use HTML fragments or JSON[5] in P1 arose due to the limits of HTML. For example, to partially update a web page and change the content of an HTML table, the two choices were to either programmatically manipulate the DOM or to directly replace the table element with an updated table. An HTML table element simply has no underlying data model which can be edited and allowing for the table element to be re-rendered when the data in the model is changed.

Initially, the first approach was used, having the web server returning a JSON string which was evaluated to a JavaScript object. Using the MochiKit DOM library, the data in the object was then used to construct the table rows. Updating the table this way required a fair amount of client-side scripting and eventually the second approach was favoured instead. Generating HTML code is typically something web frameworks do well, why it was easier to have WebWork generate the HTML code for the table and return it to the client. A simple script was then used to replace the old table with the updated one.

Form elements in HTML, such as input fields, are fairly easy to update after the web page has been rendered in the web browser, why JSON was still used as the data transport format in the implementation of the form auto-complete features.

The limitation of not being able to create underlying data models to be used by the user interface elements does not apply to Swing-based applications. Using Swing's JTable component as an example, a custom table model can be created to provide the user interface component with data. When the data model is altered, the table component can be refreshed to display the most recent data.

[5] JavaScript Object Notation (JSON) is a data-interchange format based on a subset of JavaScript (URL 7).

6.1.2 Type conversion errors did not short-circuit validation

One of the features of WebWork is type conversion of form data. The problem was that when a type conversion failed, a validation error was displayed along with the error message from the failed type conversion. This was considered a problem, since it does not make much sense to validate data if it is not of the correct type.

When using validation in WebWork, it is possible to short-circuit the validation rules so that only one validation error gets displayed per incorrect field. It would have been desirable to have the type conversion error reporting work the same way. If the conversion fails, only the type conversion error gets displayed.

For example, suppose that the user enters a string value in a field where an integer value greater than ten is expected. This would cause a type conversion error to be displayed, but the validation constraint would also cause a second error message to be displayed, telling the user that the entered value is less than ten. Since the entered value is not even an integer, this feedback is probably of little value to the user.

A question on how to solve the issue was asked in the WebWork support forum. The proposed solution was to disable the *conversionError* interceptor for the action and instead use a custom conversion validator for the field. Although this solution works, it requires the programmer to add conversion validators to each field in the application for which conversion errors are to be displayed.

6.1.3 JavaScript Hijacking

JavaScript Hijacking, also known as cross-site request forgery, is a security vulnerability that may arise when using a data transport format where the messages themselves are valid JavaScript statements. As JSON by definition is a subset of JavaScript, web applications using JSON can be vulnerable.

If a user visits a malicious website, an attack to another website which uses JSON for transferring data can be initiated. By making clever use of scripting, typically by overloading constructor functions, the malicious website can gain access to confidential data, assuming the user is logged in on the website under attack.

The malicious website uses a script-tag to issue a request to the website under attack. When the browser evaluates the JSON response, the overloaded constructors intercept object creation and allow for scripts from the malicious website to gain access to the data.

In P1, JSON was used to transfer hotel guest information, which means the way it was implemented made it directly vulnerable to JavaScript Hijacking. To protect web applications against this vulnerability, the application can be designed so that the server declines malicious requests. Preventing the JSON response from being directly executed by the browser is another way to protect the application. This can be achieved by for example enclosing the response in a JavaScript block comment.

As JavaScript Hijacking is technique for bypassing browser security policies, standalone Java clients such as P2 are not vulnerable to this specific kind of attacks. Nevertheless, it is of course important to take the necessary security measures also when developing stand-alone clients.

JavaScript Hijacking and how to protect a web application against it has been discussed by Chess et al. (2007).

6.2 Problems with the second prototype

As the second prototype was only a proof-of-concept implementation and therefore very limited in scope, there were no direct problems apart from a couple of minor configuration errors. Nevertheless, in a real-world application of greater extent than the prototype, there will likely be drawbacks also with the JWS approach and potential problems are discussed in chapter six.

6.3 Comparison based on user experience qualities

To show the pros and cons of the two approaches from a user experience point of view, these have been compared based on the qualities described by Mullet (2003).

6.3.1 Seamlessness

Comparing the flow and responsiveness of web applications and standalone desktop applications is a bit like comparing apples and oranges – they are run under quite different prerequisites. Web applications often suffer from low responsiveness since a full-page reload typically has to be made each time the user makes a new request. By using AJAX techniques, the responsiveness can be increased by making partial page updates. But still, unless the application is designed to run in a single page, even though AJAX is utilised there is a need for full-page reloads as the user navigates through the application.

One benefit of web applications is the few steps it takes the first time the application is launched. Typically launching is done by the user clicking a link on a web page or entering a URL in the browser to reach the application's start page. Once this page is loaded, the user can start interacting with the application immediately. This means the entry barrier for starting to use a web application is low.

Although not necessarily more complex, launching an application using Java Web Start the first time requires a couple of more steps, resulting in a less seamless user experience. First, the user commonly clicks a link on a web page pointing at the application's JNLP file. The next step varies slightly depending on how the user's browser is configured, but typically the user will be asked what to do with the JNLP file and is given the option to open it using JWS – assuming JWS is installed. The files necessary for running the application will then be downloaded and depending on the size of the application, this may take a while. After all the required downloads are complete, the user will be prompted whether to accept the application publisher's certificate or not – unless the publisher is already a trusted source. Finally, the actual application is launched.

For subsequent launches of the application, the user experience is likely to be more seamless, since JWS uses cached versions of the required application resources unless they have been updated. Also, the possibility to add a desktop shortcut to the application makes it unnecessary to start the application via the web browser.

About the responsiveness in Swing applications, the Swing toolkit suffers from an old reputation of being slow. This reputation stems to a large degree from incorrect usage of the toolkit, related to Swing being single-threaded and not thread safe. In many applications, long running tasks are invoked from GUI event handlers, which are executed on Swing's event dispatch thread (EDT). When a long running task is invoked on the EDT, the GUI freezes until the task has finished executing. Correct usage of Swing is instead to dispatch long running tasks to worker threads, thereby avoiding blocking of the EDT, to keep the GUI responsive. In Java 6.0, the SwingWorker class was introduced to make it easier to dispatch lengthy tasks to their own threads.

Due to the fact that Swing is not thread safe, it is also important that all interaction with Swing components takes place only on the EDT. Two utility methods exist for this purpose, `SwingUtilities.invokeLater()` and `SwingUtilities.invokeAndWait()`. When the GUI state is to be manipulated, a `Runnable` with code invoking methods on the Swing components is passed to one of the utility methods. The code is then ensured to be executed on the EDT.

These are the two main rules to avoid threading issues when developing Swing applications (Marinilli, 2006):

- Do not perform long running tasks on the event dispatch thread.
- Manipulate Swing components only from the event dispatch thread.

In order to have a responsive user interface, it is important to follow these rules when developing a Swing application. Naturally, there can also be other causes for an application being slow, but adhering to the above rules is a good start.

6.3.2 Focus

That the application is focused is much up to how the user interface is designed and not so dependant of the underlying technologies. Yet, one thing that can be said about web applications in general is that since they are run in a web browser, they are applications running inside another application. This can impose some problems, making the user experience less focused.

When using a web application, the browser's user interface is typically visible and shortcut commands used in the browser can affect how the web application behaves. One example of how the browser's behaviour can affect the behaviour of a web application is the backspace key, which in many browsers is a shortcut for navigating to the previously visited page. In a form-based web application, the user expects to be able to erase data entered in a form by using the backspace button. If however, for some reason none of the form elements is focused when the user hits backspace, the browser will navigate to the previous page, which was not the intention of the user.

Another consequence of having the application running in the browser is that users are likely to expect the browser features to work as normal. For example being able to add bookmarks and use the navigation buttons. This can be a problem when using AJAX, especially if the application runs in a single page. When an AJAX application makes a partial page update, it uses the XML HTTP request object, meaning that from the browser's point of view the user is still visiting the same page and the URL will not change. Users expect clicking the back button will take them to the previous *state* of the application, but unless the application has been designed with back button issues in mind, the browser will navigate to the previous actual *page*. There are ways to solve this problem, see for example Almaer et al. (2006) for more information.

6.3.3 Connectedness

The two prototypes were both designed as service-based back-end applications, but using two different front-end technologies. For this reason, they both offered the same possibilities to connect to remote services as this is typically handled in the back-end layer. Instead, the major difference regarding connectedness was in the capabilities of being able to design the application to work in an offline or occasionally connected mode.

To function properly offline, any non-trivial application will need to be able to access pre-fetched resources and queue updates to be performed once the application is back online. The limitations of web applications make this hard to achieve unless additional browser extensions, such as Google Gears[6], are used. The only standard way for web applications to store state locally is by using cookies, which do not allow for storage of large amounts of information for offline processing.

An application deployed using JWS on the other hand, given the permission of the user, can access the local file system. This means resources can be downloaded while the application is online, allowing for it to remain functioning when taken offline. For example, an online word processing application with a centralised storage repository could allow the user to edit documents while being offline. Once the application has network access again, the edited files can be synchronised with the central repository.

Another use case is an application which is completely standalone and does not perform any remote communication at all. Such an application could still benefit from being deployed using JWS to leverage its automatic update feature. Since JWS caches applications and their resources, it would be possible to run the application without any network connection, assuming it has been configured to allow for offline usage. If the application or its resources are updated, the latest versions will be downloaded the next time the application is started with the network available.

6.3.4 Awareness

For an application to be aware it needs to be able recognise the context in which it is currently operating. One such feature, that is supported by both WebWork and standalone Java clients, is determination of the user's locale to allow for localisation of the application. Leveraging the locale information, the application can be adapted to display for example time and currencies in a way that is familiar to the user.

[6] Google Gears is an open source browser extension that enables web applications to provide offline functionality (URL 5).

Mullet (2003) means that context-sensitive adaptation has never been a strength of interactive software and apart from localisation, the technologies used for the two prototypes provide no specific features out of the box for creating adaptive interfaces. Nevertheless, adaptive interfaces can still be created, but the implementation is up to the application designers themselves. Input fields with auto-complete functionality, as commonly seen in AJAX-enhanced web applications, is an example of a feature that was implemented in both of the prototypes. By using basic knowledge of what information the user is supposed to enter, the user's task can be simplified by partly automating the input process.

7 Discussion

7.1 Difficulties during the study

During the development of the prototype front-end, no major problems were encountered and the web-based technologies showed to be sufficient for implementing the user interface. One possible reason why the limitations of web-based applications never became apparent is probably because it was decided already at an early stage that the front-end was to be implemented as a web application. This fact was then, unconsciously or not, taken into account during the initial phases of the project – requirements gathering and design. As the purpose of this thesis was to study how limitations and problems related to web applications could be addressed by instead using stand-alone client technologies and Java Web Start, it became problematic as the initial prototype did not reveal any major limitations of the web technologies that were used. For this reason it was decided to also compare the two approaches based on user experience qualities.

7.2 Justification of the user experience quality comparison

If two diverse implementations of an application technically solve the same problem, however using different technologies, then I strongly believe providing a better user experience can be an important success factor. Another justification for comparing the technologies based on user experience qualities is that in the era of "Web 2.0" there have been a lot of discussions of how to create "richer" web applications (see for example O'Rourke, 2004 or Schwerin, 2006). By making the user experience quality comparison, my hope is to shed some light on what the term "richer" actually means in the context of user interface development.

One can argue that it is not fair to make a comparison based on user experience qualities identified by the Macromedia Experience Design team, due to the fact that Macromedia – now owned by Adobe – was and remains being one of the major vendors in the rich internet application development space. Personally I do not consider this a problem, since in this study only Java-based technologies have been compared of which none have been developed by Macromedia or Adobe.

7.3 Sufficiency of Java Web Start

Two of the reasons why web applications initially became popular were due to low deployment costs and the easiness with which applications could be updated from a single location. The aim of the Java Web Start technology is to bring these benefits to the desktop platform by letting the software vendor publish the application and its resources online. As new versions of the software are released, Java Web Start automatically downloads the updates to the client.

The way the update feature of Java Web Start works is that each time the application is started, a check is performed to make sure that all resources are up-to-date. If any of the resource files has been updated since the last time the application was launched, the updated files are downloaded before the application is started. But what will happen if the application is restarted only rarely?

One simple scenario where the update feature of Java Web Start is not sufficient is if the hotel staff by the end of the day, instead of turning off the computer on which the application is running, just lock the screen when leaving their working shift without exiting the application. The next day the hotel staff unlock the screen and start using the computer again. Following the same pattern, the application will never be restarted, meaning no update checks will be performed.

A front office application such as the hotel property management system will likely be one of the central software artefacts among the applications used by hotel staff. For such an extensive application, a more advanced update feature might be needed. If it can not be assumed that the application is restarted on a regular basis, it would be better if updates could be downloaded transparently while the application is running. Once the download is complete, the application could notify the user and if needed ask the user to restart the application. Some of the available rich client platforms – frameworks for desktop client development – offer this functionality. An interesting future extension of the work done in this thesis would be to rewrite the second prototype to leverage one of these platforms. For example, the NetBeans RCP has an auto-update module that could possibly solve the problem described above (URL 15).

For web applications, the same scenario would not cause any troubles. Instead, since they are hosted on central servers, web applications are automatically kept up-to-date.

7.4 Current limitations of web browsers and future enhancements

One major drawback of standard web applications is that they require a working internet connection. For web applications to partly function offline, they would need to be able to store resources locally. Local storage is one of the features of the recently released beta version of the web browser extension Google Gears. Looking at the features of Google Gears highlights some of the current limitations of web applications.

Apart from the module allowing for local storage, Google Gears also includes a local relational database module and a WorkerPool module. The database module uses the SQLite (URL 18) database and exposes a JavaScript API for storing and retrieving data by executing SQL statements.

The WorkerPool module addresses a web browser problem, analogous to the aforementioned problem with long-running tasks executed on the Swing EDT (see 6.3.1), where resource-intensive scripts can cause the user interface to become unresponsive. By allowing web applications to run scripts in the background, the WorkerPool module aims to solve this problem.

Another project that aims to improve the web browsing experience is Tamarin (URL 14). The project was started in November 2006 when Adobe contributed the source code for their ActionScript virtual machine used in the Adobe Flash Player (URL 13). Tamarin is hosted as an open source project by the Mozilla Foundation and the goal is to implement a high-performance virtual machine for the ECMAScript 4th edition. Currently it implements the 3rd edition of the ECMAScript standard, the standard on which both JavaScript and ActionScript are based. The virtual machine has several performance features and includes a just-in-time compiler for faster code execution. It will be used within SpiderMonkey, the JavaScript engine embedded in Mozilla-based products, such as Firefox. The Tamarin code will also continue to be used by Adobe within the Flash Player. There are also plans to enable for other browsers to use the virtual machine.

The described browser improvements are not yet standard. The beta version of Google Gears is available for download, but the Tamarin project is as of writing not included in any of the Mozilla applications. Needless to say, I believe these improvements will blur the gap between the web and desktop applications even more.

7.5 Future enhancements of client-side Java

During the JavaOne conference 2007, Sun announced that they are currently working on an update of the Java runtime environment called the "Consumer JRE" (URL 7). The update is planned to be released during the first half of 2008 and will introduce new features addressing some of the existing problems with the JRE (ibid).

The task of determining if a user already has a JRE installed on her machine can be troublesome (see 2.2). With the Deployment Toolkit, included in the Consumer JRE, this problem will be addressed, by providing developers with an interface to detect and – if necessary – install Java on the user's machine. Also, the update mechanism for users who already have Java installed will be enhanced.

One complaint often heard is that the JRE download is too big, compared to for example the Flash Player. The Consumer JRE will address this issue with a feature called the Kernel installation mode, which will allow for downloading parts of the JRE incrementally as they are needed. While this feature only enhances the experience for first time Java users, it is an improvement that would make the installation process more seamless.

Another complaint is that it takes too long when first starting a Java application. This is because it takes time to load all the needed resources into memory when the Java virtual machine is first launched after a system reboot. To overcome this issue, typically referred to as cold-start, the Consumer JRE will introduce a feature called the Quick Starter, which pre-fetches parts of the JRE into memory and leverages the operating system cache.

The Consumer JRE is part of the user experience improvements Sun are trying accomplish with their JavaFX initiative (URL 9). Part of JavaFX is also JavaFX Script, a new language specifically designed for graphics and animation, which runs in the JRE on the desktop. I personally think the JavaFX initiative shows that Sun is now putting more effort into desktop Java.

8 Conclusions

The initial prototype of the hotel property management system did not reveal any major limitations of the web technologies that were used. However, by comparing the two prototypes based on user experience qualities, differences between them started to show.

A stand-alone desktop client typically offers higher responsiveness than a web application and therefore makes it easier to create a *seamless* user experience. Running outside the context of the web browser can make the experience more *focused*. With local file system access, a stand-alone desktop application can be designed to degrade nicely if the connection to the back-end is lost. Such a feature could improve the redundancy of the application and enhance the *connectedness* user experience. The final user experience quality in the comparison, *awareness*, is less dependant on the technologies being used.

The major problem with Java on the desktop is to make sure that the user has the proper version of the Java runtime environment installed. With the improvements offered by the upcoming Consumer JRE, the extent of this problem will likely be reduced.

The Java Web Start technology provides an easy way for deploying stand-alone desktop clients and keeping them automatically updated. However, for an application such as the hotel property management system, I believe a more advanced update feature will be needed. Relying on the application to be updated based on that the user restarts the application regularly might not be sufficient. Still, for such a large and complex application as I believe a full-scale hotel property management system would be, I think implementing the application as a stand-alone client is an interesting alternative.

Initially, web applications can often be easy to implement, but only to a certain degree. The more features that are packed into a web application, the more the browser starts resembling a runtime environment. Why not then instead use software that has been designed from the ground-up to be an application runtime environment?

Glossary

Central Reservation System	A software system used within a chain of hotels for processing reservations (URL 24).
Global Distribution System	A software system which links bookers, such as travel agencies, to travel suppliers' booking systems (URL 25).
Property Management System	A computerised system for integrating all elements of hospitality information and management (URL 26).

List of acronyms

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CRS	Central Reservation System
CSS	Cascading Style Sheets
CRUD	Create, Read, Update and Delete
DAO	Data Access Object
DHTML	Dynamic HTML
DOM	Document Object Model
EDIFACT	Electronic Data Interchange For Administration, Commerce and Transport
EDT	Event Dispatch Thread
FDS	Flex Data Services
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IIS	Internet Information Services
JAR	Java Archive
JNLP	Java Network Launch Protocol
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
JWS	Java Web Start
MVC	Model View Controller
MXML	(no official meaning given)
OTA	OpenTravel Alliance
PMS	Property Management System
RCP	Rich Client Platform
RPC	Remote Procedure Call
RIA	Rich Internet Application
SOA	Service-Oriented Architecture
SQL	Structured Query Language
UML	Unified Modelling Language
WPF	Windows Presentation Foundation
XAML	eXtensible Application Markup Language
XHTML	eXtensible Hypertext Markup Language
XML	eXtensible Markup Language
XUL	XML User Interface Language

References

Literature

Almaer, Dion, Galbraith, Ben & Gehtland, Justin (2006). *Pragmatic* AJAX *– A Web 2.0 Primer.* Pragmatic Bookshelf

Alur, Deepak; Crupi, John & Malks, Dan (2001). *Core J2EE Patterns: Best Practices and Design Strategies.* Pearson Education, first edition

Andersson, Ulrika; Lundquist, Malin; Merkel, Magnus & Önnegren, Britta (2006). *Lathund för rapportskrivning.* Linköpings universitet

Bates, Bert & Sierra, Kathy (2005). *Head First Java.* O'Reilly, second edition

Brown, Charles E. (2007). *The Essential Guide to Flex 2 with ActionScript 3.0.* Friends of ED

Carreira, Jason & Lightbody, Patrick (2005). *WebWork in Action.* Manning

Cederholm, Dan (2004). *Web Standards Solutions – The Markup and Style Handbook.* Friends of ED

Cederholm, Dan (2006). *Bulletproof Web Design – Improving Flexibility and Protecting Against Worst-Case Scenarios with* XHTML *and* CSS. New Riders

Chess, Brian, Tsipenyuk O'Neil, Yekaterina & West, Jacob (2007). *JavaScript Hijacking.* Fortify Software

Garrett, Jesse James (2002). *The Elements of User Experience – User-Centered Design for the Web.* New Riders

Handy, Alex (2005). *Where Are the Rich Internet Applications Written in Java?* Software Development Times, issue 139, BZ Media

Heinemeier Hansson, David & Thomas, Dave (2005). *Agile Web Development with Rails.* Pragmatic Bookshelf

Holzschlag, Molly & Shea, Dave (2005). *The Zen of* CSS *Design – Visual Enlightenment for the Web.* New Riders

Kabanov, Jevgeni & Mürk, Oleg (2006). *Aranea – Web Framework Construction and Integration Kit.* ACM International Conference Proceedings Series; vol. 178, pp. 163 – 172, ACM Press

Keith, Jeremy (2007). *Bulletproof AJAX.* New Riders

Lindvall, Mikael (1997). *An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution.* Linköping Studies in Science and Technology, Dissertation No. 480

Loosley, Chris (2006). *Rich Internet Applications: Design, Measurements and Management Challenges.* Keynote Whitepaper

Marinilli, Mauro (2006). *Professional Java User Interfaces.* Wiley

Meyer, Eric A. (2004). *CSS Pocket Reference – Visual Presentation for the Web.* O'Reilly

Mills, Duncan (2003). *Understanding MVC.* ODTUG Technical Journal; June 2003 issue

Mullet, Kevin (2003). *The Essence of Effective Rich Internet Applications.* Macromedia White Paper

Næssén, Olof (2007). *Utvärdering av webbramverk för Java.* LITH-IDA-EX—07/034—SE

O'Rourke, Cameron (2004). *A Look at Rich Internet Applications.* Oracle Magazine; vol. 18, no. 4, pp. 59 – 60

Plaisant, Catherine & Shneiderman, Ben (2005). *Designing the User Interface – Strategies for Human-Computer Interaction.* Addison Wesley, fourth edition

Schwerin, Rich (2006). *Getting Rich with AJAX.* Oracle Magazine; vol. ?, no. 5, pp. 65 – 66

Weiss, Aaron (2005). *WebOS: Say Goodbye to Desktop Applications.* NetWorker; vol. 9, issue 4, pp. 18 – 26, ACM

Yin, Robert K. (2003). *Case Study Research Design and Methods.* Sage Publications, third edition

Zeldman, Jeffrey (2003). *Designing with Web Standards.* New Riders

Internet

1. Adobe (May 2nd 2007). *Adobe – Flex 2 – Application Development.* http://www.adobe.com/products/flex/

2. Adobe (Oct 10th 2007). *Adobe Labs – Air.* http://labs.adobe.com/technologies/air/

3. Amadeus (June 5th 2007). *Amadeus Press Kit.* http://www.amadeus.com/amadeus/x67283.html

4. Apache (May 3rd 2007). *Struts – Welcome.* http://struts.apache.org

5. Google (Sep. 25th 2007). *Google Gears (BETA).* http://gears.google.com

6. Haase, Chet; Ng, Thomas & Nourie, Dana (Aug. 25th 2007). *Auto-Install: Easier Launching of Java Web Start Applications.* http://java.sun.com/developer/technicalArticles/JavaLP/javawebstart/AutoInstall.html

7. Haase, Chet & Laux, Thorsten (Oct. 7th 2007). *JavaOne 2007 – Desktop Java Technology Today.*
http://developers.sun.com/learning/javaoneonline/2007/pdf/TS-3160.pdf

8. JSON.org (Oct 8th 2007) *JSON in Java.*
http://www.json.org/java/

9. Marinacci, Joshua (Oct. 7th). *JavaFX != JavaFX Script.*
http://weblogs.java.net/blog/joshy/archive/2007/09/javafx_javafx_s.html

10. MochiKit (May 3rd 2007). *MochiKit Makes JavaScript Suck Less.*
http://www.mochikit.com

11. Microsoft (May 12th 2007). *Microsoft Silverlight – Light Up the Web.*
http://www.microsoft.com/silverlight/

12. Mozilla (May 11th 2007). *The Joy of XUL – MDC.*
http://developer.mozilla.org/en/docs/The_Joy_of_XUL

13. Mozilla (Sep. 27th 2007). *Adobe and Mozilla Foundation to Open Source Flash Player Scripting Engine.*
http://en.www.mozilla.com/en/press/mozilla-2006-11-07.html

14. Mozilla (Oct. 8th 2007). *Tamarin Project.*
http://www.mozilla.org/projects/tamarin/

15. Netbeans (Oct. 9th 2007). *Autoupdate: netbeans.org : autoupdate.*
http://autoupdate.netbeans.org

16. OpenLaszlo (May 3rd 2007). *Software Engineer's Guide to Developing OpenLaszlo Applications.*
http://www.openlaszlo.org/lps/docs/guide/

17. OpenTravel Alliance (Jan. 26th 2007). *OpenTravel Alliance.*
http://www.opentravel.org

18. SQLite (Oct. 8th). *SQLite home page.*
http://www.sqlite.org

19. Sun Microsystems (June 11th 2007). *Java Web Start version 1.5.0 – Frequently Asked Questions*
http://java.sun.com/j2se/1.5.0/docs/guide/javaws/developersguide/faq.html

20. Sun Microsystems (Aug. 3rd 2007). *JNLP File Syntax.*
http://java.sun.com/j2se/1.5.0/docs/guide/javaws/developersguide/syntax.html

21. Wikipedia (Jan. 26th 2007). EDIFACT.
http://en.wikipedia.org/wiki/EDIFACT

22. Williams, Kurt (Dec. 15th 2006). *Mismatch Mish-Mash - Cardsharp on Software*.
http://www.jroller.com/page/cardsharp?entry=mismatch_mish_mash

23. XULPlanet (May 11th 2007). *Why Use XUL?*
http://www.xulplanet.com/tutorials/whyxul.html

24. CatererSearch (Dec. 6th 2007). *Technology Jargon Buster*.
http://www.caterersearch.com/Articles/2005/08/31/302345/technology-jargon-buster.htm

25. John Beech (Dec. 6th 2007). *Glossary*.
http://www.stile.coventry.ac.uk/cbs/staff/beech/BOTM/Glossary.htm

26. Pearson Education (Dec. 6th 2007). *Glossary*.
http://wps.pearsoned.co.uk/wps/media/objects/1881/1926829/glossary/glossary.html

Appendix A – Use cases

Guest check-in

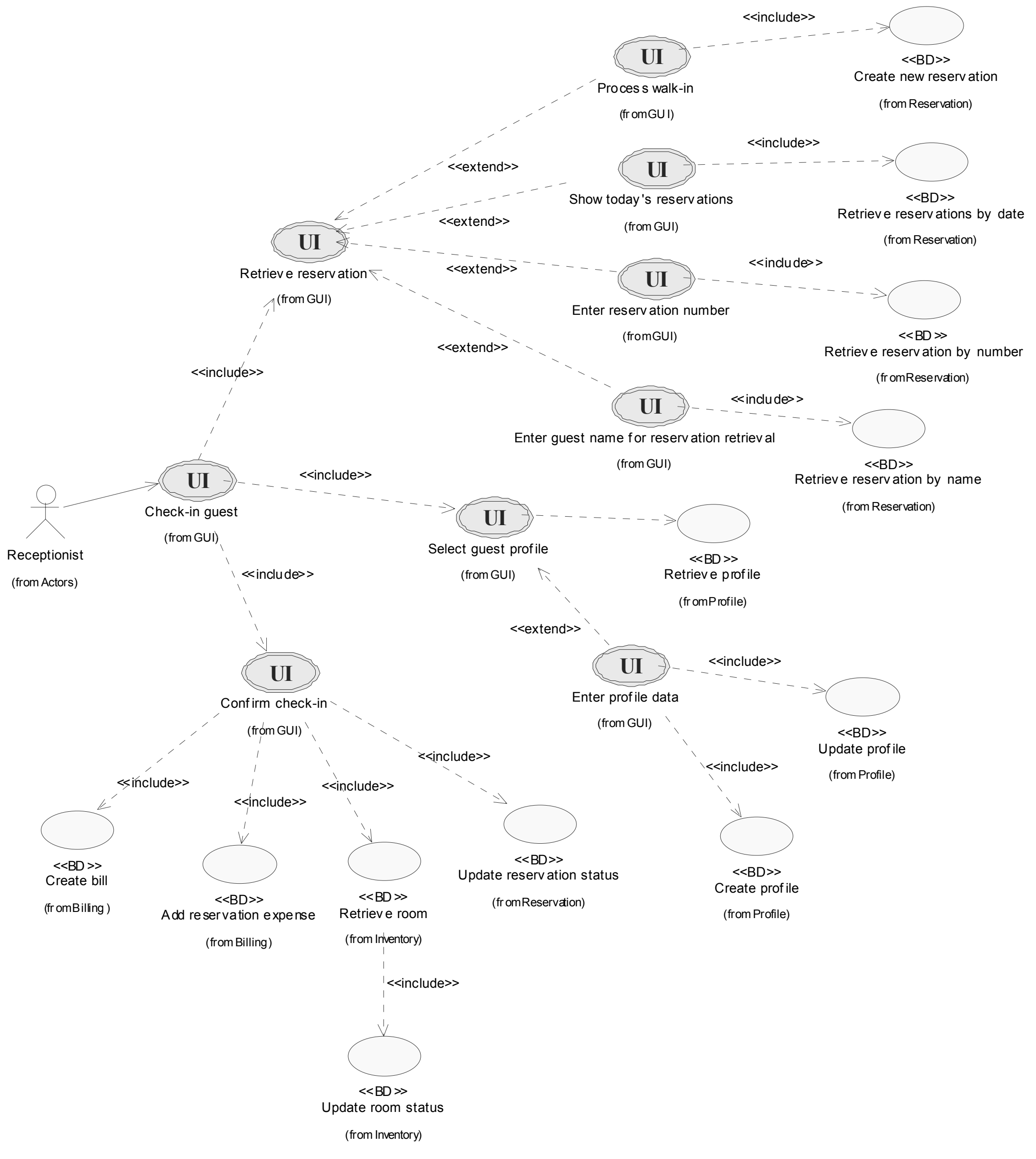

Figure 5. Central Property Management System :: Check-in

Guest check-out

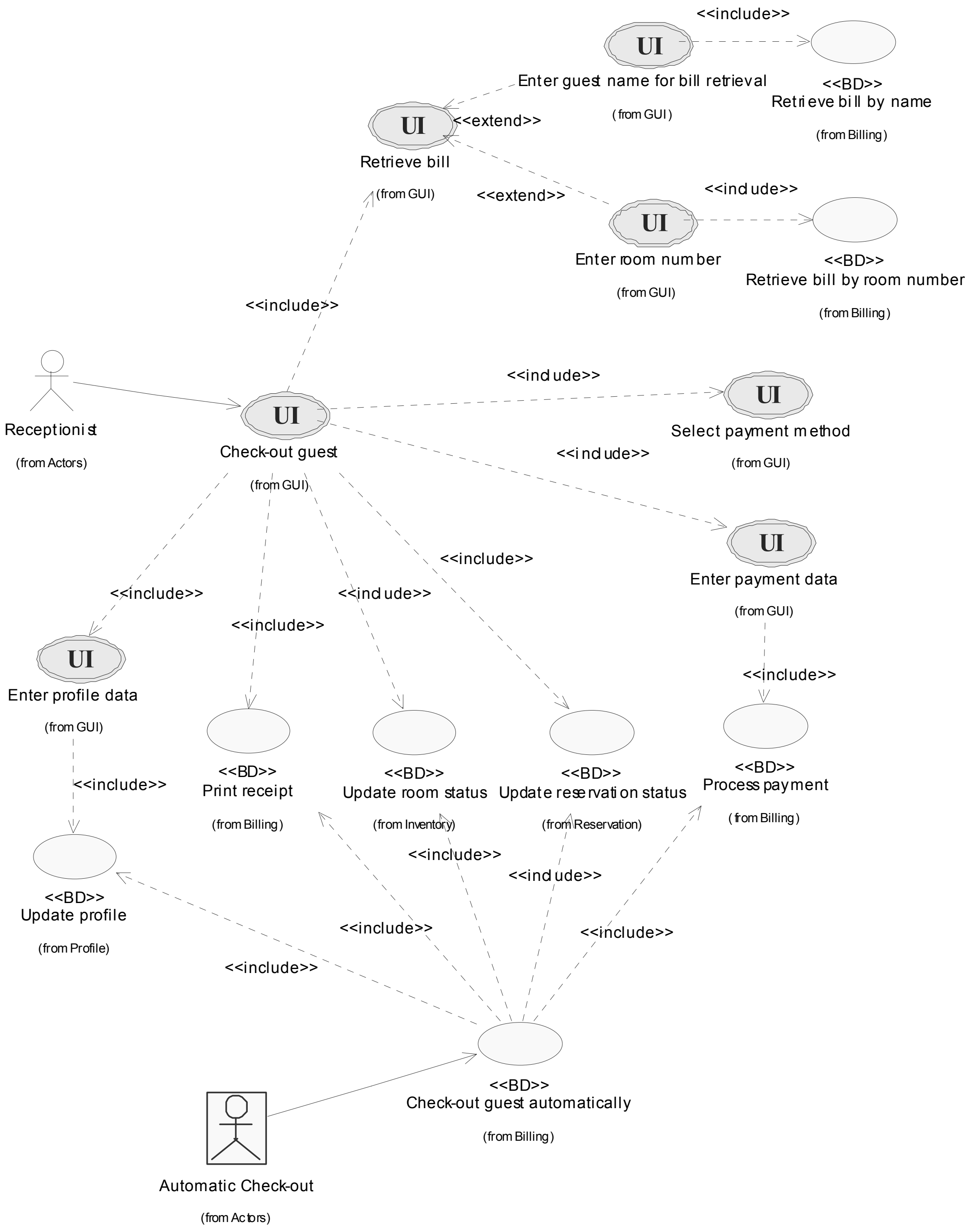

Figure 6. Central Property Management System :: Check-out

Appendix B – WebWork sample application

The source code files listed constitute a localised WebWork application with form validation. First listed is the `xwork.xml` file, which is where each action is mapped to view templates and an action class. The Java property files provide translated strings necessary for enabling localisation of the application. Then there are two JSP template files, leveraging WebWork's tag library and defining the application's user interface. The last two files are a validation metadata file used by the validation framework and a Java action class. In this example the same action class is used for both of the application's actions.

This sample application can be run in a standard servlet container, for example Tomcat or Jetty, complete with the WebWork library and its dependencies.

config/xwork.xml

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork 1.1.1//EN"
 "http://www.opensymphony.com/xwork/xwork-1.1.1.dtd">

<xwork>
  <include file="webwork-default.xml" />
  <package name="default" extends="webwork-default">
    <action name="form" class="greetapp.HelloAction">
      <result name="success">form.jsp</result>
    </action>
    <action name="greet" class="greetapp.HelloAction" method="greet">
      <result name="input">form.jsp</result>
      <result name="success">greet.jsp</result>
    </action>
  </package>
</xwork>
```

config/GlobalMessageResources_en.properties

```
greet.title = WebWork sample application
greet.form.label = Name
greet.form.button = Send
greet.form.validationErrorMessage = You must enter a name
greet.message = Hello {0}. Current time is {1}
```

config/GlobalMessageResources_sv.properties

```
greet.title = WebWork-exempel
greet.form.label = Namn
greet.form.button = Skicka
greet.form.validationErrorMessage = Du måste ange ett namn
greet.message = Hej {0}! Klockan är {1}
```

config/webwork.properties

```
webwork.i18n.encoding=UTF-8
webwork.custom.i18n.resources=GlobalMessageResources
```

webapp/form.jsp

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<%@ taglib uri="/webwork" prefix="ww" %>

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

<head>
   title><ww:text name="greet.title"/></title>
</head>

<body>

  <h1><ww:text name="greet.title"/></h1>

  <ww:form action="greet" method="post">

    <ww:textfield name="userName"
        label="%{getText('greet.form.label')}"/>

    <ww:submit value="%{getText('greet.form.button')}" />

  </ww:form>

</body>

</html>
```

webapp/greet.jsp

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<%@ taglib uri="/webwork" prefix="ww" %>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title><ww:text name="greet.title"/></title>
</head>
<body>
  <h1><ww:text name="greet.title"/></h1>
  <p>
    <ww:text name="greet.message">
      <ww:param value="userName" />
      <ww:param value="time" />
    </ww:text>
  </p>
</body>
</html>
```

java/greetapp/HelloAction-greet-validation.xml

```
<!DOCTYPE validators PUBLIC
 "-//OpenSymphony Group//XWork Validator 1.0.2//EN"
 "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">

<validators>

  <field name="userName">
    <field-validator type="requiredstring">
      <message key="greetapp.form.validationErrorMessage" />
    </field-validator>
  </field>

</validators>
```

java/greetapp/HelloAction.java

```
package greetapp;

import java.text.DateFormat;
import java.util.Calendar;

import com.opensymphony.xwork.ActionSupport;

public class HelloAction extends ActionSupport {

    private String userName;


    public String greet() {
        return SUCCESS;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getUserName() {
        return userName;
    }

    public String getTime() {
        final Calendar calendar = Calendar.getInstance();
        final DateFormat df = DateFormat.getTimeInstance();

        return df.format(calendar.getTime());
    }

}
```

Appendix C – JNLP-file for the Java Web Start application

The file below is the Java Network Launching Protocol (JNLP) file which was used for the second implementation of the prototype. It shows some of the available parameters for configuring Java Web Start.

HotelPMS.jnlp

```
<?xml version="1.0" encoding="UTF-8"?>

<jnlp spec="1.0" codebase=http://localhost/jws/
    href="HotelPMS.jnlp">

    <information>
        <title>Hotel Property Management</title>
        <vendor>Emil Jönsson</vendor>
        <homepage href="index.html"/>
        <description>A prototype of a Hotel PMS</description>
        <icon href="icon.png"/>
        <icon href="splash.jpg" kind="splash"/>
        <offline-allowed/>
    </information>

    <resources>
        <j2se version="1.5+"/>
        <jar href="HotelPMS-0.1.1.jar"/>
        <jar href="log4j-1.2.9.jar"/>
    </resources>

    <application-desc
        main-class="pms.jws.prototype.HotelApplication"/>

</jnlp>
```

Appendix D – Internship timetable

2006

September 4th	Start of internship Requirements gathering phase
September 26th	Product Specification document finished Design phase started
October 16th	High Level Design document finished Implementation phase started
December 22nd	Front-end implemented

2007

January 8th	OTA XML message design Study of back-end architecture
February 9th	Implementation of back-end services started Integration of front-end and back-end
March 2nd	Internship finished