Reactive User Interfaces in the Web

M.Sc. Thesis
University of Turku
Department of Information Technology
Computer Science
2015
Johannes Dahlström

Supervisors:
First Supervisor
Second Supervisor

Contents

Chapter 1

Introduction

Interactive software systems have become progressively more complex, driven by the demand for richer user interaction along with the growing multimedia capabilities of modern hardware. Applications are increasingly being written for the World Wide Web platform, and the inherently distributed nature of web applications brings forth its own challenges. Not only are they required to serve an ever-increasing number of users, but also, more and more commonly, to facilitate interaction *between* users.

Complex interaction requires complex user interfaces. In object-oriented programming, the usual means of implementing a user interface is to make use of the Observer design pattern. In this pattern, interested objects can register themselves as *observers*, or listeners, of events, such as mouse clicks or key presses, sent by user interface elements. When an observer is notified of an event, it reacts in an appropriate manner by initiating a computation or otherwise changing the application state. Similarly, the user interface may react to events triggered by some underlying computation, signaling the user that something requires his or her attention.

The traditional observer pattern has several disadvantages. Observers are difficult to compose and reuse, leading to instances of partial or complete code duplication. Events often lack important context, making it necessary to manually keep track of the program state and share the bookkeeping between different observers. The resulting code typically forms a complex and fragile state machine, making it difficult to reason about its behavior and correctness.

In the recent years, several mainstream object-oriented programming languages used in the industry have adopted concepts traditionally belonging to the relatively academic realm of functional programming. These include functions as first-class values; anonymous functions (lambdas); and combinators such as map, filter, and reduce. One impetus for this paradigm shift has been the constantly increasing importance of concurrency and parallelism in software. Correctly managing and reasoning about mutable state shared between concurrent threads of execution is notoriously difficult, and the general disposition towards immutable state in functional programming has proven to be a useful basis for building better concurrency abstractions.

Reactive programming is a programming style centered on the concept of propagating change. In a reactive system, a variable can be bound to other variables so that its value changes automatically as a response to value changes in other components of the bound system. A common example of such reactivity is a spreadsheet application, where the value of a cell can be a formula referring to several other cells. The displayed value of the cell is refreshed whenever the value of a referenced cell changes. Another name for this approach is dataflow programming.

One way of improving upon the observer pattern is to elevate the abstract concept of an event stream to a first-class data type. Event streams, or *observables*, can then be merged, transformed, and otherwise manipulated in a declarative manner using the familiar set of functional tools.

Vaadin is a web application framework written in Java, aiming to provide a rich set of user interface components facilitating rapid application development. It also contains a data binding layer for propagating input and output between the user and a data model. Both the user interface and data binding are designed in terms of the traditional observer pattern.

This thesis seeks to answer the question of whether reactive programming techniques are useful in writing user interfaces in Vaadin. Furthermore, it seeks to analyze whether some of these techniques should be adopted by Vaadin itself instead of simply being built on top of the framework.

Chapter 2 provides an overview of the history of the Web as an application platform and the current state of the art of Web applications. Chapter 3 introduces the reactive programming paradigm and discusses its usage in the context of Web applications. In Chapter 4, a reactive framework, built on top of Vaadin, is presented. The viability and potential for further development is analyzed in Chapter 5, and Chapter 6 provides conclusions.

Chapter 2

Programming Web Applications

2.1 A Brief Introduction to the Web

The World Wide Web, or the Web for short, is a massive distributed information system in the Internet. It constitutes a large set of interlinked *hypertext documents*, accessed using a browser application.

The Web project was initiated by the British computer scientist Timothy Berners-Lee while working at the European Organization for Nuclear Research, or CERN, in the late 1980s and early 1990s. His original aim was to improve information sharing among scientists in the nuclear research community, but the Web soon expanded to more general use. [1] [2]

Hypertext, one of the central concepts of the Web, refers to electronic documents linked to other documents via embedded references called hyperlinks. An early vision of the concept was presented in the seminal 1945 essay *As We May Think* by Vannevar Bush [3]. In the 1960s, the term “hypertext” was coined by Ted Nelson, and a working hypertext system was showcased by Douglas Engelbart in the famous ”Mother of All Demos”.

Hypertext documents in the Web are written in HTML (Hypertext Markup Language), which is based on the older SGML (Standardized General Markup Language). An HTML document is a structured text file consisting of a hierarchy of nested elements, representing the logical and visual structure of the document. A Web browser interprets the HTML and renders a graphical representation of it to the user, managing layouting, typesetting, multimedia, and possible interactivity as specified in the document. A simple HTML page is shown in Listing 2.1. [4]

Listing 2.1: A small HTML document.

```html
<!-- Displays a traditional greeting -->
<!DOCTYPE html>
<html>
  <head>
    <title>Greetings</title>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <h1>Hello World!</h1>
    <p>This is a traditional greeting.</p>
  </body>
</html>
```

The Web is based on a client-server architecture utilizing the Hypertext Transfer Protocol (HTTP). A client application, typically a Web browser, connects to an HTTP server, requesting a *resource* such as a Web page, an image, or other type of file. Resources are identified via unique textual identifiers, URIs. [5]

A single server can attend to several clients—constrained by the available resources—but the clients cannot communicate directly with each other. The server must work as a mediator in any client-to-client interaction. Another limitation in the HTTP model is that the server cannot proactively send messages to the clients; it may only respond to requests initiated by them. Furthermore, the protocol is stateless; two consecutive requests by the same user are independent and not associated with the same session without separate bookkeeping.

A Web resource need not be a physical file served from mass storage; the server may instead elect to generate the response partially or fully programmatically. Thus, when a user reloads a Web page, its content may change dynamically without manual maintenance. For instance, the server might do a database query and present the results to the client as a formatted HTML document. In practice, it is useful to modularize a Web server so that it can delegate request handling to a set of subprograms on a request-by-request basis.

A simple protocol, *Common Gateway Interface* (CGI), was developed to facilitate such delegation from a Web server to an auxiliary program. Early CGI applications were typically written in C or Perl. For each request, the server would execute the associated CGI application as a separate process, passing it details of the request in environment variables. The application would write an HTTP response to its standard output stream and the server would send the response to the client. [6]

In the early days of the Web, the only form of interaction between the user and the Web server was requesting pages either by typing an explicit URI or following hyperlinks. Use cases soon emerged for the ability for the user to send input data to the server; the latter in turn serving another page based on the user input and possibly save the input to persistent storage for future use. In 1993, Mosaic, one of the first graphical browsers, added to its HTML dialect a rudimentary set of input elements, including text fields, buttons, and list boxes. These elements were later included in the HTML 2 standard [4].

Compared to regular desktop applications, this rudimentary interactivity was rather slow and awkward. To process any user input, the browser would have to send a HTTP request to the server, where it would be processed and a new Web page, generated based on the input, served to the client. Fetching up-to-date content from the server would require the user to manually ask the browser to refresh the page.

To achieve more fulfilling interaction, a client-side programming model was necessary. In 1995, Brendan Eich developed the first version of *JavaScript*, an interpreted language executed by the browser. The language could be used to dynamically manipulate the document, typically interactively as a response to input events such as mouse clicks and key presses. For security reasons, JavaScript code in a browser is executed in a ”sandbox”, a secure virtual machine, and the language initially offered practically no access to standard operating system services such as the file system or the network. Partially due to these limitations, client-side scripting was considered by many a novelty at best and a nuisance at worst. Listing 2.2 exhibits a simple JavaScript program that displays a pop-up dialog as a response to a button click.

Listing 2.2: A small JavaScript program.

```
var btn = document.getElementById("button");

btn.addEventListener("click", function () {
   alert("Clicked a button!");
});
```

To better utilize the distributed aspect of the Web in client-side programming, a method for making programmatic HTML requests, without necessitating the reloading of the whole page—and losing all client-side state—was required. Such functionality was developed by Microsoft in the late 1990s and became known as *Ajax* (Asynchronous JavaScript and XML). It is arguable that Ajax was the technology that started the ongoing age of Web applications.

As the name implies, Ajax requests are *asynchronous*: the browser initiates the request in the background and notifies the script of its eventual completion. As a security measure, Ajax connections can, by default, only be made to the server from which the JavaScript code itself was requested.

HTML 5 is a common name for various technologies that aim to improve the capabilities and richness of interaction of Web applications. Several of these new features are programming interfaces that expose operating system services to JavaScript in a regulated fashion, including persistent storage, networking, and accelerated 3D graphics. [7]

The Web has been gradually transformed from a simple document retrieval system to a full-fledged distributed application platform. Despite their potentially awkward user interfaces, Web applications have several advantages. They do not require a separate installation step, they run on any platform with a modern Web browser, and they are intrinsically network-aware, permitting interaction with not just the server, but also with other users connected to the same server. While historically user interactions in the Web may have had a latency of several seconds, with modern Web technologies even highly interactive applications such as fast-paced multi-player computer games are feasible. [1]

2.2 Web Application Architecture

In short, a Web application is a program accessed with a Web browser or a specialized HTTP client. HTML pages, either computer-generated or hand-written, are used to present a graphical interface to the user, and HTTP requests are utilized to transmit information between the client and the server as necessary. JavaScript is used to provide low-latency interactivity in the browser, and there has been a constant push to move more and more application logic to the client side. Consequently, there has been a rising demand to improve the performance and capabilities of JavaScript programs, and browser vendors have responded to that demand.

[1]As an anecdote, this thesis was written in part using the SHARELaTeX service at `http://www.sharelatex.com`.

2.2.1 Communication

HTTP, the protocol underlying the Web, is *request-oriented* and *stateless*. For the original use case of document retrieval this was more than sufficient, but issues arise when it is attempted to use as the communication protocol of a distributed application.

As a request-oriented, or request-response protocol, HTTP connections have a transient nature. A client establishes a network connection to the server and sends a request message specifying the URL of the resource it wants to fetch, as well as auxiliary information in the form of request headers. The server procures the requested resource, provided one is available, and sends it back to the client along with response headers. After this, the request is complete and the connection closed. Listings 2.3 and 2.4 give examples of an HTTP request and response, respectively. [5]

Listing 2.3: A typical HTTP request.

```
GET /index.html HTTP/1.1
Host: www.example.com
Connection: keep-alive
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
    image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (
    KHTML, like Gecko) Chrome/30.0.1599.66 Safari/537.36
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
```

Besides not keeping a connection open, by default an HTTP server does not even keep track of whether two separate requests belong to the same logical user session. Indeed, in the traditional CGI approach, the server would spawn a new instance of the CGI program for each handled request; the instance would execute, generate a response, and then halt. Any request-to-request state would have to be saved in a persistent storage such as the file system or a database.

Session tracking between requests can be implemented using *cookies*, short textual data coupled with an expiration date that are passed between the server and the client. A cookie sent by the server is stored by the client and included in all subsequent requests to the same server until it expires. Simple state can be encoded directly in the cookie, but a more common approach is to persist session state on the server side and simply share a session identifier with the client. [8]

Listing 2.4: A typical HTTP response.

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Type: text/html
Content-Length: 45
Server: Apache 2.4.1

<!DOCTYPE html>
<html>
  <body>
    <h1>Hello</h1>
  </body>
</html>
```

The bulk of the client-server communication in a modern Web application is done using Ajax requests. Before Ajax, the only ways to do a HTTP request in a browser were navigating to a URL either by entering it manually or clicking a link, or submitting an HTML form; the latter would typically load a new page as well. A full page load disrupts user interaction and loading and rendering the page takes time. Furthermore, the state of any JavaScript programs on the page is lost. With Ajax, it is possible to communicate with the server in the background, maintaining interactivity and the state of the user interface.

Because JavaScript programs are single-threaded, an Ajax request cannot simply block until the request is complete without preventing user interaction during that time. Instead, a *callback* function is associated with the request, invoked by the browser at certain stages of the request handling. In most cases, only the last stage—response received—is of interest. Listing 2.5 demonstrates the making of an Ajax request and its asynchronous completion.

An important limitation of Ajax requests is that they are subject to a security measure called the *same-origin policy*. This prevents a JavaScript program from connecting to servers other than the one from which the script was loaded. Cross-origin resource sharing (CORS) is a mechanism developed to selectively loosen this restriction.

Since it is not possible for an HTTP server to initiate communication with the client, *polling* must be used if the client wishes to keep informed of possible changes occurring on the server. Often this is done manually by the user; sometimes JavaScript or special HTML tags would be used to reload the page periodically. A sudden unexpected reload

Listing 2.5: Making an Ajax request.

```
var request = new XMLHttpRequest();
request.onreadystatechange = function() {
   if(request.readystate == 4) {
      // Response received
      if(request.status == 200) {
         handleResponse(request.response);
      } else {
         signalFailure();
      }
   }
};
request.open("GET", "http://www.example.com");
request.send(); // Returns immediately
```

will typically not make the user very happy, however. Ajax requests can be used to poll the server and partially update the page contents without overly distracting the user. If state changes on the server are irregular, frequent polling may lead to a large number of superfluous requests. On the other hand, polling more infrequently leads to increased latency in delivering the updates to the client.

Several mechanisms have been developed to facilitate a more flexible way of server-to-client communication, often called *server push*. As opposed to the client *pulling* content from the server, the server is pushing data to the client when needed. Some push mechanisms utilize HTTP requests that are purposely kept open for longer than what is typical; the response is not sent until the server wishes to notify the client of some event. These techniques are, out of necessity, somewhat haphazard and *ad hoc* in nature, and may not work reliably in an environment that expects HTTP requests to behave in a traditional manner.

WebSocket is a recent technology, part of the HTML 5 family of Web standards, aiming to improve the networking capabilities of Web applications. In particular it can be used to implement an efficient and reliable server push mechanism. WebSocket connections are persistent, bidirectional, and impose considerably less overhead than HTTP. They are also not subject to the same-origin policy.

2.2.2 Single-page Web Applications

Traditional Web sites are composed of multiple pages, between which the user navigates via hyperlinks. In contrast, many modern Web applications have embraced a single-page architecture, where most or all content changes are done programmatically by manipulating a single HTML page. These are called, fittingly enough, *single-page Web applications*.

-Initial page fully created on server, or bootstrapped with JS

-Rich Internet Applications - rich JavaScript UIs

-UI events cause Ajax requests

- Server replies with page fragments or instructions controlling JS

2.3 User Interface Programming

- Text-based vs graphical

- interactive applications react to input, typically wait most of the time

The computer mouse and first mouse-based interfaces were developed by a team led by Douglas Engelbart at the Stanford Research Institute in the 1960s. Graphical user interfaces were pioneered by Xerox in the 1970s.

-Mobile interfaces: touch instead of mouse, small screens

2.3.1 Widgets

A typical graphical user interface consists of a visual hierarchy of rectangular display elements, variously called controls, components, or *widgets*. Widgets are usually reusable; their behavior can be customized so that, for instance, clicking two different buttons may have entirely distinct effects.

Widgets can be loosely classified into three types: *layout widgets*, *output widgets*, and *input widgets*. Layout widgets are used to visually group other widgets in order to visually indicate to the user their significance and logical relationships. Layouts can typically be nested. Output widgets are concerned with presenting data—textual, numeric, audiovisual or other—and input widgets allow user to enter input data.

A core set of widget types has existed since the days of the Xerox Alto. These include windows, labels, push buttons, dropdown menus, and list boxes. (...)

2.3.2 Events and Observers

-Event loop

-Observer pattern [9]

-Listing 2.6

Listing 2.6: An implementation of the observer pattern.

```
interface Subject {
   void addObserver(Observer);
   void removeObserver(Observer);
}

interface Observer {
   void notify();
}

class ConcreteSubject implements Subject {
   private Collection<Observer> observers =
      new ArrayList<>();

   public void addObserver(Observer o) {
      observers.add(o);
   }

   public void removeObserver(Observer o) {
      observers.remove(o);
   }

   protected void notifyObservers() {
      for(Observer o : observers) {
         o.notify();
      }
   }
}
```

2.3.3 Model-View-Controller

In adherence to the well-established design principles of modularity and separation of concerns, when programming interactive applications it is often desirable to make a clear distinction between the user interface and the underlying data model manipulated via that interface. A common way to structure an application in this manner is the *Model-View-*

Controller (MVC) pattern. It divides the architecture of an application into three distinct layers, each having well-defined responsibilities:

- The Model layer is concerned with the data model, its integrity, constraints and validation of data, and its storage into and retrieval from an underlying persistence mechanism such as a relational database.
- The View layer implements the user interface, fetching data from the model and displaying it to the user. (...)
- Finally, the Controller concerns itself with handling user input, modifying the model and changing the view appropriately. (...)

There are several variations of MVC, such as Model-View-Adapter and Model-View-Presenter. In the former, instead of the view directly fetching data from the model, the controller (now called adaptor) mediates all interaction between the other two layers. In the latter, user interaction logic is pushed from the view to a presenter layer, making the view itself simply a passive conduit of input and output.

A closely related concept in client-server programming is *multitier architecture*. (...)

In a Web application, each of the layers can straddle the client-server division. (...)

2.4 Vaadin

Vaadin is a Web application framework written in Java that aims to make the design and maintenance of high-quality Web-based user interfaces easy. It attempts to abstract away many of the more inconvenient aspects of the Web platform, trying to provide an experience similar to traditional desktop application programming.

-Client access is there if needed

-Recognized that the abstraction is leaky

-Write server-side UI code, client side rendered automatically

-Events and updates transmitted automatically

2.4.1 Architecture

-GWT

-Compile Java to JavaScript

Listing 2.7: A simple Vaadin application.

```java
public class HelloUI extends UI {

   protected void init(VaadinRequest request) {
      VerticalLayout layout = new VerticalLayout();
      Button button = new Button("Click me!", clickEvent -> {
         Notification.show("Clicked!");
      });
      layout.addComponent(button);
      setContent(layout);
   }
}
```

-Partial implementation of Java Standard Library in addition to Java APIs for JS/DOM stuff

-Clientside counterparts for server-side components created and maintained automatically

2.4.2 Communication

-Ajax or push

-Shared state and RPC

-Widgets have state that can only be changed by the server, changes sent to client

-Client can make remote procedure calls to server to notify about events, request more data etc

-Server can make RPCs to client

2.4.3 Component Model

-Fairly standard collection of widgets, called components

-Basic observer pattern for event handling

2.4.4 Data Model

-Bind fields to data

-Data source can be memory, DB, Web service, ...

-Container, Item, Property

-”Smart” wrappers for data

-Bidirectional propagation of change

-Based on observer pattern

-Converters

-Validators

-Transactionality

Chapter 3

Reactive Programming

3.1 Reactive Primitives

3.1.1 Behaviors and Events

3.1.2 Futures and Promises

3.1.3 Observables

3.1.4 Async and Await

3.1.5 Actors

3.1.6 Composability

3.2 Reactive libraries

3.2.1 Rx

3.2.2 React.js

Chapter 4

Case Study: Reactive Vaadin

Chapter 5

Validation

Chapter 6

Conclusions

References

[1] T. Berners-Lee. Information Management: A Proposal. CERN internal proposal, March 1989.

[2] T. Berners-Lee and R. Cailliau. WorldWideWeb: Proposal for a HyperText Project. CERN internal proposal, November 1990.

[3] V. Bush. As We May Think. *The Atlantic Monthly*, 176(1):101–108, July 1945.

[4] T. Berners-Lee and D. Connolly. Hypertext Markup Language – 2.0. RFC 1866, RFC Editor, November 1995.

[5] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. J. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, RFC Editor, June 1999.

[6] D. Robinson and K. Coar. The Common Gateway Interface (CGI) Version 1.1. RFC 3875, RFC Editor, October 2004.

[7] I. Hickson, R. Berjon, S. Faulkner, T. Leithead, E.D. Navara, E. O'Connor, and S. Pfeiffer. HTML5: A vocabulary and associated APIs for HTML and XHTML. W3C Proposed Recommendation, W3C, September 2014.

[8] A. Barth and U.C. Berkeley. HTTP State Management Mechanism. RFC 6265, RFC Editor, April 2011.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, October 1994.

[10] R. Cailliau. A Little History of the World Wide Web, 1995.

[11] Anne van Kesteren. XMLHttpRequest Level 2. W3C working draft, W3C, January 2012.

[12] ECMAScript® Language Specification. Standard ECMA-262 Edition 5.1, Ecma International, June 2011.

[13] A. Le Hors, P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document Object Model (DOM) Level 3 Core Specification. W3C Recommendation, W3C, April 2004.

[14] M. Grönroos. *Book of Vaadin*. Vaadin Ltd, Vaadin 7 edition, April 2013.

[15] D. Coward and Y. Yoshida. Java^{TM} Servlet 2.4 Specification. JSR 154, Sun Microsystems, Inc., November 2003.

[16] R. Mordani. Java^{TM} Servlet Specification, version 3.0. JSR 315, Sun Microsystems, Inc., December 2009.

[17] A. Leff and J.T. Rayfield. Web-application development using the Model/View/-Controller design pattern. In *Enterprise Distributed Object Computing Conference, 2001. EDOC '01. Proceedings. Fifth IEEE International*, pages 118–127, 2001.

[18] J. Conallen. Modeling Web Application Architectures with UML. *Commun. ACM*, 42(10):63–70, October 1999.

[19] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[20] M. Jazayeri. Some Trends in Web Application Development. In *Future of Software Engineering, 2007. FOSE '07*, pages 199–213, May 2007.

[21] L. Shklar and R. Rosen. *Web Application Architecture: Principles, Protocols and Practices*. Wiley, 2nd edition, April 2009.

[22] Microsoft Patterns & Practices Team. *Microsoft® Application Architecture Guide (Patterns & Practices)*. Microsoft Press, 2nd edition, November 2009.

[23] M. Galli, R. Soares, and I. Oeschger. Inner-browsing extending the browser navigation paradigm, May 2003.