Reactive User Interfaces with Vaadin

Johannes Dahlström

Thesis for Degree of Master of Science
Computer Science
Department of Information Technology
University of Turku
2016

Supervisors:

TURUN YLIOPISTO
Informaatioteknologian laitos

JOHANNES DAHLSTRÖM: Reactive User Interfaces with Vaadin

Pro gradu, 28 s., 0 liites.
Tietojenkäsittelytiede
(Kuukausi) 2016

Suomenkielinen tiivistelmä

Asiasanat: Vaadin, web, reaktiivinen ohjelmointi, käyttöliittymät

UNIVERSITY OF TURKU
Department of Information Technology

JOHANNES DAHLSTRÖM: Reactive User Interfaces with Vaadin

Thesis for Degree of Master of Science, 28 p., 0 app. p.
Computer Science
(Month) 2016

The English abstract

Keywords: Vaadin, web, reactive programming, user interfaces

Contents

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 carries 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 (lambda expressions); and combinators such as map, filter, and reduce. An 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.

A possible 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.

2 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]

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.

Listing 2.1: A small HTML document.

```
<!-- 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>
```

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 a set of 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 store 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 an HTTP request to the server, where it would be processed and a new Web page, generated based on the input, was then 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.

```
1 var btn = document.getElementById("button");
2
3 btn.addEventListener("click", function () {
4    alert("Clicked a button!");
5 });
```

To better utilize the distributed aspect of the Web in client-side programming, a method was required for making programmatic HTML requests, without necessitating the reloading of the whole page and losing all client-side state. 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.

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 produces 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.

```
1 GET /index.html HTTP/1.1
2 Host: www.example.com
3 Connection: keep-alive
4 Cache-Control: max-age=0
5 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/
     webp,*/*;q=0.8
6 User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML,
     like Gecko) Chrome/30.0.1599.66 Safari/537.36
7 Accept-Encoding: gzip,deflate,sdch
8 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

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

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>
```

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]

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.

When Ajax was initially conceived, JavaScript programs were strictly single-threaded.[2] Consequently, a request could not simply block until its completion 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

[2]This restriction has recently been relaxed via the introduction of the WebWorkers mechanism.

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.

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
```

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 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. Moreover, they are 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 the document tree of a single HTML page. These are called, fittingly enough, *single-page Web applications*.

The page on which the application operates may constitute a fully functional initial view, assembled by the server, or it may contain the minimal HTML and JavaScript code required for "bootstrapping" the client side.

-Rich Internet Applications - rich JavaScript UIs

-UI events cause Ajax requests

- Server replies with page fragments or instructions controlling JS

2.2.3 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 specific 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 component, 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. (...)

2.3 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 the 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 [3]

2.4 Concurrency

2.4.1 Asynchronous Input and Output

[3] Java listings use Java 8 features, such as lambda expressions, whenever appropriate.

Listing 2.6: An implementation of the observer pattern in Java 8.

```
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() {
      observers.forEach(o -> o.notify());
   }
}
```

3 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

Listing 3.1: A simple Vaadin application.

```
1  public class HelloUI extends UI {
2
3      protected void init(VaadinRequest request) {
4          VerticalLayout layout = new VerticalLayout();
5          Button button = new Button("Click me!", clickEvent -> {
6              // Display a floating notification
7              // each time the button is clicked
8              Notification.show("Clicked!");
9          });
10         layout.addComponent(button);
11         setContent(layout);
12     }
13 }
```

3.1 Background

The origins of Vaadin trace back to the year 2000 when a group of experienced Web developers based in Turku founded the company IT Mill. The objective of the team was to build a tool that would

-Millstone

In 2006, the toolkit had a major rewrite as the client-server communication was changed to use AJAX requests

-IT Mill Toolkit 5

Toolkit version 5, released at the end of 2007, contained a completely rewritten client side based on Google Web Toolkit (GWT). GWT is a set of libraries and a Java-to-JavaScript compiler (or "transpiler") that allows writing browser-based applications in plain Java. This enables experienced server-side Java programmers to also be productive writing client-side code when needed.

-Vaadin 6

IT Mill Toolkit was re-branded as Vaadin in 2009. Shortly thereafter the company itself was also renamed Vaadin for branding purposes.

-Vaadin 7

Vaadin 7 was again a major upgrade that modernized many aspects of the framework, removing legacy code and revamping the communication layer. The client-side architecture was also greatly improved by logically separating UI components (widgets) from the client-server communication logic, allowing the widgets, in principle, to be used in pure GWT applications.

Unless otherwise specified, all information in the subsequent sections pertains to Vaadin 7.5, the most recent non-maintenance release at the time of writing.

3.2 Architecture

3.2.1 Google Web Toolkit

Google Web Toolkit, or GWT for short, is a Web technology that enables writing browser-based applications in Java instead of JavaScript. It constitutes a set of Java libraries wrapping native JavaScript APIs, a partial implementation of the Java standard library, and a compiler for translating Java to JavaScript. It was originally developed by Google for internal use; the first public release was in May 2006. GWT is licenced under the Apache 2 licence.

The GWT

UI components in GWT are called *widgets*.

3.2.2 Communication

Vaadin is based on the Servlet technology, the standard for writing web applications in Java. Each

-ajax or push

By default all communication between the client and the server occurs over HTTP. Asynchronous (AJAX) requests are used to transmit notifications of user actions to the server and to convey consequent changes to the application state back to the client. There are two primary mechanisms of communication provided for components: *shared state* and RPC *invocations*. Shared state is component state that should be known to both the client and the server, and synchronized whenever the server-side state changes. It is read-only on the client. RPC (remote procedure call)

is a mechanism for invoking selected methods on the other endpoint over the network. Its main use is notifying the server of client-side events, but the server can also make RPC calls to the client in cases it feels more natural than using the shared state.

Both shared state and RPC invocations are serialized and transmitted as JSON strings.

3.2.3 Component Model

User interface components in Vaadin are called simply *components*. Every component implements the interface `com.vaadin.Component`.

The root component containing all other components in a Vaadin application instance is called a UI, represented by class `com.vaadin.UI`. A Vaadin UI may encompass a complete Web page or be embedded within content not controlled by Vaadin. A user may have a Vaadin application open in multiple browser windows concurrently; each corresponds to a separate UI instance.

Vaadin has a straightforward implementation of the Observer pattern to handle user interaction. The observers are called *(event) listeners* and the subjects *(event) notifiers*.

Each Vaadin server-side component class has a widget counterpart on the client side. Each component also has a corresponding *connector* class, used as a bridge between the client-side widget and the server-side component class. The connector typically listens to events emitted by the widget and notifying the server accordingly. When the state of the server-side component changes, the connector receives a notification and synchronizes the widget state. This makes it possible to adapt widgets designed for pure GWT for Vaadin use, and vice versa - use Vaadin widgets in plain client-side GWT applications.

When a component is added to the UI on the server side, the change is automatically communicated to the client. The client creates an instance of the corresponding connector class, which in turn is responsible for creating a widget of a correct type.

3.2.4 Data Model

-”Smart” wrappers for data

-Bind fields to data

In addition to setting and querying the value of Vaadin field components explicitly, any field instance may be assigned a *data source*, an instance of interface `com.vaadin.data.Property`. It is an object that wraps a value and notifies any registered listeners whenever the value is changed.

-Container, Item, Property

A container constitutes a set of items, and an item is composed of a set of properties.

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

The Container abstraction is designed to hide the underlying storage mechanism - it might be a Java `Collection` object, a relational database

-Validators

-Bidirectional propagation of change

-Transactionality

Fields also propagate user input back to their data source. The input is first passed through zero or more *validators*, all of which must agree that the value is valid before it can be committed. If the input does not validate, any validation errors are reported to the user. If a field is *buffered*, its value must be explicitly committed to the data source. Otherwise, the value is propagated automatically.

-Based on observer pattern

-Converters

Fields and properties are parameterized by their value type. If the value type of a field and its data source differs, a *converter* can be used to convert values between the types.

4 Reactive Programming

Reactive programming refers to a class of related programming patterns that are centered on the concept of propagating change.

4.1 Functional Programming Basics

Functional programming is a programming paradigm that XXX.

4.1.1 Purity and immutability

A function is said to be *pure* if it has no side effects and its return value only depends on its argument values. A very desirable property of pure functions is *referential transparency*: any expression involving only calls to pure functions can always be replaced by the evaluated value of the expression independent of its context.

4.1.2 Lists

The most fundamental data structure in functional programming is the immutable *list*. A list is typically recursively defined as either the *empty list* $\emptyset$ or a pair $(x : xs)$ composed of a value x (the *head* of the list) and another list xs (the *tail* of the list). The list constructor $(\cdot : \cdot)$ is generally understood to be right associative (4.1).

$$(x_1, x_2, x_3, \ldots, x_n) \equiv x_1 : x_2 : x_3 : \cdots : x_n : \emptyset \equiv x_1 : (x_2 : (x_3 : \cdots : (x_n : \emptyset))) \tag{4.1}$$

The recursive structure of a list makes it straightforward to recursively traverse.

4.1.3 Lazy evaluation

4.1.4 Pattern matching

4.1.5 Monads

A *monad* is a structure that, in a sense, abstracts out the concept of sequencing, or chaining, computations on values in pure functional programming.

The concept of a monad was originally adopted from category theory to pure functional programming language Haskell to represent I/O while keeping the language pure and free of side effects.

Formally, a monad is a type with a type constructor and two associated operations, *unit* (4.2) and *bind* (4.3). [1]

$$\text{unit} :: t \rightarrow M(t) \tag{4.2}$$

$$\text{bind} :: (M(t), t \rightarrow M(u)) \rightarrow M(u) \tag{4.3}$$

The unit operation simply wraps a value in an instance of the monadic type.

-Operations: bind and return

-Alternatively: flatten and map $\rightarrow$ flatMap

In addition, for a monad have the expected semantcs, the definitions of unit and bind must obey certain axioms. These identities are usually called the *monad laws*.[2]

$$\text{bind}(\text{unit}(x), f) \equiv f(x) \tag{4.4}$$

$$\text{bind}(m, \text{unit}) \equiv m \tag{4.5}$$

$$\text{bind}(\text{bind}(m, f), g) \equiv (x) \rightarrow bind(f(x), g) \tag{4.6}$$

The Maybe Monad

Many programming languages have the concept of a *null value*. The value null is a placeholder signifying “no value”; typically trying to use null where a value is expected results in an error raised by the runtime.

Perhaps the simplest useful example of a monad is the *Maybe type*, also known as *Option* or *Optional*. It represents an optional value: a container that may either contain a single value or

[1] The symbol :: should be read “has type”.

[2] The $\equiv$ symbol should be read “is semantically equivalent to”.

nothing at all. These two cases are often called *Some* and *None*, respectively. The bind and unit operations are defined in 4.7 and 4.8.

$$\text{unit}(x) = \text{Some}(x) \tag{4.7}$$

$$\begin{aligned}&\text{bind}(m, f) = m \textbf{ match}\\ &\quad \textbf{case } \text{None} \Rightarrow \text{None}\\ &\quad \textbf{case } \text{Some}(x) \Rightarrow f(x)\end{aligned} \tag{4.8}$$

The `bind` operation allows us to compose functions that take plain values and return Maybes

$$\begin{aligned}&x \textbf{ match}\\ &\quad \textbf{case } None \Rightarrow None\\ &\quad \textbf{case } Some(x) \Rightarrow foo(x) \textbf{ match}\\ &\quad\quad \textbf{case } \text{None} \Rightarrow None\\ &\quad\quad \textbf{case } \text{Some}(x) \Rightarrow bar(x)\\ &\quad\quad\quad \textbf{case } \text{None} \Rightarrow None\\ &\quad\quad\quad \textbf{case } \text{Some}(x) \Rightarrow baz(x)\end{aligned} \tag{4.9}$$

$$\text{bind}(\text{bind}(\text{bind}(x, \text{foo}), \text{bar}), \text{baz}) \tag{4.10}$$

$$x \gg= \text{foo} \gg= \text{bar} \gg= \text{baz} \tag{4.11}$$

```
for {
   y <- foo(x)
   z <- bar(y)
   w <- baz(z)
} yield w

x.flatmap(foo).flatmap(bar).flatmap(baz)
```

The List Monad

The list data structure introduced in subsection 4.1.2 also forms a monad with definitions 4.12 and 4.13.

$$\text{unit}(x) = [x] \tag{4.12}$$

$$\begin{aligned}&\text{bind}(m, f) = m \textbf{ match}\\ &\quad \textbf{case } \emptyset \Rightarrow \emptyset\\ &\quad \textbf{case } x : xs \Rightarrow f(x) + \text{bind}(xs, f)\end{aligned} \tag{4.13}$$

The implementation of unit is intuitive; it simply wraps the value in a single-element list. Bind, however, is more interesting.

4.1.6 Combinators and Higher-Order Functions

A *higher-order function* is a function that either takes one or more functions as arguments or returns a function as its result. All other functions are called *first-order functions*.

Some of the most well-known higher-order functions are the list combinators `map`, `filter`, and `reduce`.

Map

The `map` combinator applies a function to each element of a list, returning a new list containing the return values (4.14).

$$\begin{aligned}&\text{map} :: ((A \to B), [A]) \to [B]\\ &\text{map}(f, \emptyset) = \emptyset\\ &\text{map}(f, x : xs) = f(x) : \text{map}(f, xs)\end{aligned} \tag{4.14}$$

Filter

The `filter` combinator applies a predicate (a boolean-valued function) to each list element, returning a list of only those elements for which the predicate returned `true` (4.15).

$$\begin{aligned}&\text{filter} :: ((A \to Boolean), [A]) \to [A]\\ &\text{filter}(p, \emptyset) = \emptyset\\ &\text{filter}(p, x : xs) = \textbf{ if } p(x) \textbf{ then } x : \text{filter}(p, xs) \textbf{ else } \text{filter}(p, xs)\end{aligned} \tag{4.15}$$

Fold

The `fold` function, also called `reduce` or `accumulate`, accumulates a result by recursively applying a binary function to an element and the result of reducing the rest of the list. The result of folding an empty list is an identity element passed as a parameter. This is shown in infix notation in definition 4.16.

$$\text{fold}(*, x, id) = id * x_1 * x_2 * x_3 * \cdots * x_n \tag{4.16}$$

If the operation $*$ is not associative, a distinction must be made between a *left* and *right* fold (definitions 4.17 and 4.18 respectively).

$$\text{foldl}(*, x, id) = ((((id * x_1) * x_2) * x_3) * \cdots * x_n) \tag{4.17}$$

$$\text{foldr}(*, x, id) = (x_1 * (x_2 * (x_3 * \cdots * (x_n * id)))) \tag{4.18}$$

The right fold operation is a straightforward application of recursion (4.19).

$$\begin{aligned} &\text{foldr} :: ((A, B) \to B, [A], B) \to B \\ &\text{foldr}(f, \emptyset, id) = id \\ &\text{foldr}(f, x : xs, id) = f(x, \text{foldr}(f, xs, id)) \end{aligned} \tag{4.19}$$

The left fold, on the other hand, is somewhat trickier since XXX. In (4.20) we use the `id` argument as an *accumulator*:

$$\begin{aligned} &\text{foldl} :: ((B, A) \to B, [A], B) \to B \\ &\text{foldl}(f, \emptyset, id) = id \\ &\text{foldl}(f, x : xs, id) = \text{foldl}(f, xs, F(id, x)) \end{aligned} \tag{4.20}$$

As an aside, this implementation of foldl is *tail recursive*: the recursive invocation is the last expression to be evaluated before returning from the function. This is a desirable property since it allows a compiler to transform the recursion into an equivalent iterative loop that only requires $\mathcal{O}(1)$ stack space instead of $\mathcal{O}(n)$. The intuitive but naïve implementations of `map`, `filter`, and `foldr` above are not tail recursive; they could be turned into ones that are by delegating to a helper function with an extra accumulator parameter.

Flatmap

Another highly useful combinator is `flatmap`. It applies a list-valued function to each element of the input and then concatenates the results into a single list. From definition 4.21 we see that `flatmap` is actually the monadic bind operation of the List type!

$$\begin{aligned} &\text{flatmap} :: (A \to [B], [A]) \to [B] \\ &\text{flatmap}(f, \emptyset) = \emptyset \\ &\text{flatmap}(f, x : xs) = f(x) + \text{flatmap}(f, xs) \end{aligned} \tag{4.21}$$

Equivalences

Interestingly, the `fold` operation is in a sense more fundamental than `map`, `filter`, and `flatmap`. Each of the latter functions can be expressed in terms of the former, as demonstrated by identities 4.22, 4.23, and 4.24, but the opposite is not true.

$$\text{map}(f, x) \equiv \text{foldr}((y, ys) \to f(y) : ys, x, \emptyset) \tag{4.22}$$

$$\text{filter}(p, x) \equiv \text{foldr}((y, ys) \to \textbf{ if } p(y) \textbf{ then } y : ys \textbf{ else } ys, x, \emptyset) \tag{4.23}$$

$$\text{flatmap}(f, x) \equiv \text{foldr}((y, ys) \to f(y) + ys, x, \emptyset) \tag{4.24}$$

Other Combinators

All these combinators can be trivially implemented for the Maybe monad as well, simply by observing the analogies $\text{Some}(x) \cong [x]$ and $\text{None} \cong \emptyset$. In particular, the `flatmap` operation for Maybe is exactly the monadic bind function like it was with List. This identity generalizes to all monads; `flatmap` is simply another name for `bind`.

4.1.7 Scala

4.2 Reactive Primitives

4.2.1 Behaviors and Events

-Reactive functional programming

-Conal Elliot

-Time-varying values

-Behaviors: continuous

-Events: discrete

-Combinators

-Animation

-Push vs pull

4.2.2 Futures and Promises

A *future* is a placeholder, or a proxy, for a value that is not available when the future is created but may be at some later time, for instance, when a computation or a network request finishes. Informally, it can be seen as a (certificate of a) *promise* to either provide a value in the future or report an error in case of failure.

Some future implementations have separate future and promise types; in this case the promise is a write-only object that can be assigned a value or error exactly once (*fulfilling* the promise), and the future is a read-only view of the eventual value or error, *completed* by its associated promise. A future may be associated with several promises; in this case, only the first promise to complete the future succeeds. Similarly, a single promise may signal the completion of multiple futures.

Futures may be synchronously *waited on*; the wait operation blocks the thread of execution until the future is ready. A wait without a timeout specified may end up blocking indefinitely as there is no guarantee that the future ever completes.

-Value or error

-Typically asynchronous

-Monadic: flatmap

-More combinators: then, whenAll, whenAny, ...

4.2.3 Observables

-Stream of (future) results

-Duality

Given a type `T`,

	Pull	Push
One	`T`	`Future[T]`
Many	`Iterable[T]`	`Observable[T]`

-Combinators

-Marble diagrams

4.2.4 `Async` and `await`

-Write async code that looks synchronous

-Implemented via continuations/coroutines/resumable functions

-Compiler transformation

Listing 4.1: Async/await.

```
1 ...
```

4.2.5 Actors

-Lightweight concurrent entities

-Only communicate via message passing

-Spawn hundreds or thousands if needed

-Akka

4.3 Reactive libraries

4.3.1 Rx

4.3.2 React.js

5 Case Study: Reactive Vaadin

5.1 Refactoring the Event System

6 Validation

7 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™ Servlet 2.4 Specification. JSR 154, Sun Microsystems, Inc., November 2003.

[16] R. Mordani. Java™ 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.