UiO : Department of Informatics
University of Oslo

Applying client-side state management on server-generated web pages

Johannes Akse

Master's Thesis Spring 2016

Applying client-side state management on server-generated web pages

Johannes Akse

2016

Preface

My first encounter with the incredible concept of sharing information on a global scale, the driving force behind the World-wide web, were conveyed to me by a paperback edition of Howard Rheingolds book "Tools for thought".

Ever since, I have kept being fascinated by the seemingly unstoppable effort to continue the sharing of knowledge I have encountered in the software developer community.

At the same time I have been disappointed by the amount of websites where using the browser's navigation buttons has brought me to a different page than anticipated. Even more frustrating has been returning to a meticulously completed web form to find it wiped clean and ready to be filled out again.

Apart from the attraction toward understanding state management in a web environment, I have been wondering why there are so few academic opportunities to learn the programming language of the web.

By exploring client-side state management in my masters thesis I hope to give credibility to the JavaScript programming language also in the academic scene.

Acknowledgment

GK, Sveinung, Morten, Borris, Ivar, John, Stian, Jonas, Alle. Family and especially my beautiful, five year old daughter, who has patiently seen her father more interested in his computer than in playing.

Abstract

Managing client-side state has become increasingly complex as the web has changed from being a network for displaying static documents to the generation of dynamic content in single page applications we see today. This poses a threat to the notion of a network of uniquely identified resources as was the original intention of the World Wide Web. It often also breaks browser usability.

By implementing client-side state on a legacy website with complex server-side computations, this thesis investigate the difficulties of providing unique identifiers for dynamically generated content. Revealing some of the problems associated with accommodating to user expectations and sharing client-side state might mitigate some of the reluctance to implement the web standards.

In this thesis we look at difficulties concerning where and how to store state and issues concerning implementations of default web browser functionality like the back button, bookmarking and sharing of client state on a concrete use case. We discuss challenges that arise when applying client-side state management on top of an existing "thin-client" website with complicating server-side state logic.

The implementation is based on the Model View Controller architecture and modified to fit the web-client.

Contents

List of Figures

List of Code

Chapter 1

Introduction

> "Given that social filtering is one of the most powerful mechanisms for information discovery on the Internet, it is an utter disaster to disable the URL as an addressing mechanism" (Jacob Nielsen 1996).

1.1 Rationale

The World-wide web has made public sharing of information possible by introducing the URI as a means to identify unique resources. All the resources available client-side in a client-server relationship may comprise the state of the client. Managing this client-side state has become increasingly complex as the web has changed from being a network for displaying static documents to the web applications we see today. In the early days of the web, a document could only be in one state and it could not be changed on the client. When a new document was requested that new document became the new state. Now the client-side state is constantly changing. The user fills in a form, a list is updated by the server or a message frame pops up on screen.

This new intermingling of information, the mixing of content, pose a threat to the notion of a network of uniquely identified resources. To facilitate the sharing of information and ideas, as was the original intention of the World Wide Web, every resource needs to be distinctly identified from all others (Berners-Lee, Fielding, and Masinter 1998). All the different information available on screen at any one time and in any one location needs to be gathered into one unifying whole to be relocated and assembled on screen at a later time or at a different location. The default browser functionality of the back and forward button, the bookmarking functionality and the ability to share links to specific aggregations of content can be lost if client-side state management is not being resolved.

For websites with so-called thin clients, the URI is interpreted by the server and the resulting documents can be delivered to client computers without problem. For instance, if ordering plane tickets and a hotel room is done from a work computer you can let other travelers, your spouse or colleague, look at the hotel and other traveling specifics before ordering.

This can be done by copying the browsers address bar and send the URI to the other party.

The problem occurs when logic is moved to the client. Small calculations like choosing a higher standard of a hotel room, adding breakfast to the overall price or collecting information about means of hotel transportation will change the state of the client. Although this moving of logic to the client is a good thing—it frees both the web servers and the Internet in general from a great deal of work and is often done for performance reasons—it also demands that this state is handled in some way. Saving state on a client is not an automatic feature of web browsers and must be done by the client application developers (Castro et al. 2006).

In spite of this many big companies do not offer default browser functionality. Star Tour and Norwegian Holidays (Norwegian Air Shuttle) are examples that adhere to the above example but myriads of other large companies not conforming to the standards exist as well. For instance, webmail providers like Microsoft Exchange and Yahoo has chosen not to let users navigate application state at all. In particular, using the back button to get to a previously read email is not possible using these services.

An explanation of the lack of support for default browser functionality on big websites might have come about as a result of the declining use of some of these features. Several studies have shown that the use of the browsers default functionality has become less popular over the last 20 years (Catledge and Pitkow 1995) (Cockburn, McKenzie, and Jasonsmith 2002) (Obendorf et al. 2007) (Zhang and Zhao 2011). This can be explained by the transition from a static to a dynamic web as suggested by the over mentioned studies.

On the other hand, the decline in the use of the default browser functionality might be a result of inconsistently implemented browser functionality by application developers. If the expected functionality does not work or works in an unpredictable manner users will find other means to get to their ends (Obendorf et al. 2007).

No explicit recipes exists for remedying insufficient client-side state management. At the same time, a profusion of JavaScript frameworks and libraries address some of these problems. The HTML5 History API has also made state handling for the use on the client easier though it does not pose a solution to the problem of sharing mixed content. Neither does Fragment Identifier Messaging as it ruins the functionality of the back button (Jackson and Wang 2007).

1.2 Purpose

The purpose of this thesis is to investigate state management and the challenges encountered when applying client-side state management onto an existing website that does not offer such characteristics.

We will do this by implementing client-side state management on a concrete use case, the Genomic HyperBrowser. The use case returns server-generated documents calculated from advanced form submissions. It does

also offer mode specific representations of the calculated documents, something that adds interesting problems to the client-side state management.

In particular, we will look at difficulties concerning where and how to store state and issues concerning implementations of default web browser functionality like the back button, bookmarking and sharing of client state on the concrete use case.

The implemented state management application will be called the "StateApp".

1.3 Goals and limitations

While exploring client-side state management this thesis will address some important implementation decisions and expose pitfalls and other issues to be aware of when applying state management onto server-side legacy applications.

Other important issues related to web development, will only be elaborated if they have explicit consequences for state management.

For instance, the use of closures to produce distinct modules and thereby avoid the pollution of the global object, as advocated by JavaScript gurus like Douglas Crockford (Crockford 2008), has permeated the code base of the state inquiries done in relation to this thesis. Have these achievements come at the cost of performance? This important question is a highly interesting one yet will not be encompassed by this thesis.

1.4 Research question

To assist the exploration of what challenges arise when applying client-side state management on top of an existing "thin-client" website with complicating server-side state logic, two helper questions has been stated:

> "What challenges arise when applying state management on a thin-client website?"
>
> "What complexities are added when the existing website has complicating server-side state logic to be handled?"

1.5 Overview

Background: The background chapter will have two parts: The scientific foundations and the technical foundations.

In the first part, we will define our understanding of state and elaborate on some relevant research on the topic. We will also give a brief introduction to the controversy regarding how to separate business data from presentation and the role of the Model View Controller.

These topics will form the scientific foundations for the thesis and will act as a context for the discussion.

The motivation for the technical foundations section is to help the unfamiliar reader come up to speed on the technical matters presented in the implementation and discussion chapters. Some background on web development and design principles will be accounted for along with a short excerpt of some fundamental JavaScript concepts not found in "classical" programming. We will round up this chapter by discussing some general obstacles every web developer need to overcome.

The process of acquiring knowledge: This chapter will be used to describe the process of developing the state management application. What preparations needed to be done and what knowledge had to be acquired before starting the implementation.

Behavior-driven development, build schemes, and module based implementation will be accounted for along with notions on the implementation environment.

The finalized version of the state management application was integrated with the use case and is now an integral part of a scientific endeavor involving more than 20 informatics and biomedical researchers that are about to be submitted to a scientific journal. The integration process with this scientific environment will be presented last.

The real world use cases: This section will be started by presenting the use case and discuss some of its reported problems concerning usability.

The GSuite project, an extension of the HyperBrowser, addresses some of these usability problems but does so server side. Their solutions affect client-side state management so the aspects of the GSuite project that relate to usability will be commented on next.

Discussion: The discussion will appraise implementation specifics in the context of research presented in the scientific foundations section. The use of third-party libraries, using HTML5 API's, hijacking functionality vs passively recording state, challenges concerning the use of iframes and browser implementation inconsistencies are some of the concrete themes that will be discussed.

1.6 Justification

Managing state sounds quite simple. Just keep a central state object with the different states stored as properties and let all interested objects check the status of the central object whenever they need to synchronize.

This might be true when developing a desktop application in a vacuum where there are few external restrictions. This is not the case when adding the specifications of two legacy programs—the existing application and the web browser.

The total complexity of adding state management onto an existing application will depend on the specifications of each different project.

With a browser comes the complexity of implementing the default browser behaviors:

- A web browser has the obligations of the back and forward button
  - letting the user navigate a history list,
- the obligations of the bookmark
  - letting a user save and restore earlier state
- and the obligations of the address field
  - letting a user paste a URI sent from someone for restoring the encoded state of the URI. This also counts for adding state-full links in external documents enabling that state to be recreated in a different time and space.

This last point is also the main assertion of this thesis: that state should be stored on the URI for later retrieval such that the original idea of the World-wide web of sharing information will be maintained.

1.7 Terminology

Stringify : The process of transforming objects to strings.

Parse : The process of transforming a string to an object.

1.7.1 Acronyms

URI - Uniform Recourse Identifier.

URL - Uniform Recourse Locator.

- The octothorp. Within a URI also called the fragment identifier, the hash sign and the anchor.

API - Application Programming Interface

Chapter 2

Background

The motive for this background chapter is to create a foundation from which the rest of the thesis should be understood.

The chapter is divided into two parts. The first part introduces topics that will be prerequisite for understanding the rest of the thesis. The second part elaborates on some of the research from the introduction but also presents other relevant research and theory. This will hopefully add depth when discussing the concrete implementation done on the use case when investigating the problems stated in the introduction.

First, however, we will give an account on the notion of state.

2.1 What is state?

This last quarter of a century the web has evolved from being a network of static displays of information to the web applications we see today. Changing small parts of a single page to form highly interactive mashups of content has become the new norm. By utilizing the new XMLHttpRequest object, the client could, asynchronously, request data from a server adding information onto an already open page. This way a single web page with one URI could be displayed with completely different data. This new technique was coined "Asynchronous JavaScript and XML" (Ajax).

The "W3C technical architecture group" (TAG) consider state to be "[...] a point in some space of all possible states" (Orchard 2006). A less formal formulation of state, also stated by TAG can be: "How something is; its configuration, attributes, condition, or information content."

We understand state as the combination of all the data collected in an application from any source and the application-internal processing of this data at any one time. However, it is up to the application developer what internal data will comprise the application state and thereby what will be necessary to make available for sharing (Foster et al. 2008).

2.2 Technical foundations

This section is a courtesy to the reader with little or no knowledge of the concepts discussed in the thesis. We will try to build the foundation necessary to follow the implementation and discussion chapters.

We start by giving a short introduction to web clients and the DOM. JavaScript will also be presented and some features of the language will be explained.

We will continue this technical exploration by establishing some general ideas about design principles and the concept of "classical" design patterns. Following is a brief presentation on some issues concerning the implementation of "classical" design patterns in relation to JavaScript. These ideas and concepts will clarify the dilemmas concerning settling on a scheme for object communication discussed later in the thesis.

We will end the technical background section by introducing some general problems related to developing software in a web browser environment. Although not directly linked to the problems of client-side state management, these problems are important to be aware of when assembling a complete picture of the challenges concerning client-side state management.

2.2.1 The web browser

"The web browser is perhaps the most widely used software application in history. It has evolved significantly over the past twenty years; today, web browsers run on diverse types of hardware, from cell phones and tablet PCs to desktop computers" (Grosskurth and Godfrey 2005, p. 2). We even have iWatches and Google glasses. The web client is no longer just a reader of static information browsing interlinked web pages. Now it is better described as a simplified OS (Flanagan 2011, p.310). The browsers provide ways to organize web documents and web applications in folder structures. They allow for organization of statistical analysis questions and hypotheses. Web browsers also allow for running multiple discrete applications much like an OS. In addition, they define low-level API's for activities such as networking and saving data.

The root window object

A browser is a window for looking into all the information available on the World-wide web. The analogy of the window has been brought to the terminology of coding JavaScript in the browser as well. Upon opening a browser window the executing program creates an object called the window. This root window will be the holder of the document object when a document is requested and loaded into the browser.

The window object also holds other important properties and functionality. The "location" object holds the Uniform Resource Identifier, the information on where and how the current document was retrieved, and the "session history" object maintains a list of the recent visited documents. The

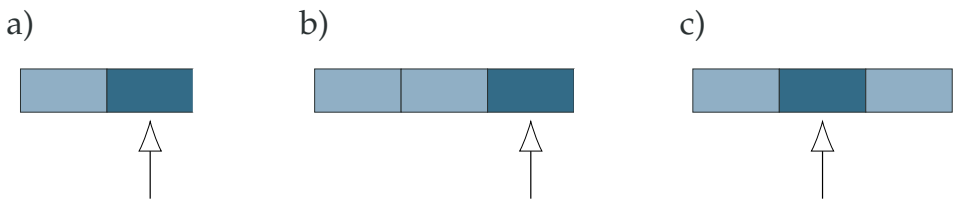


Figure 2.1: Browser session history: The arrow points at the currently displayed page. a) Base case. Two items in session history. b) Navigating to a new website. c) Navigating back to the previous page.

History object and the URI will be explained shortly, but before that we will give an account of the Document Object Model, the DOM.

The DOM

For a program to manage state within a browser context it needs some way to interact with the HTML elements displayed on the page. The standard programming language for all modern browsers, JavaScript, provides such an entry point. When the server returns the requested page the browser creates a plain JavaScript object, called the document object, and uses it as the root of a tree like structure. It then parses the received HTML. For every HTML element, it creates a responding JavaScript object and adds it to the document tree. This tree is often referred to as the DOM. All of these JavaScript objects have fields that correspond to all the HTML element attributes.

The session history object of the window element

In the early days of the World-wide web browsers displayed web pages containing static information with the possibility to link to other web pages. When clicking or otherwise navigating to another page the browser added the URL of the new page onto the session history object. This session history object was a property of the window element and contained some kind of data structure. For instance, a list, on which the visited URI's were stored. When using the browser's back button the session history list lets the window location property point to the previously added item. In this way, the browsers enabled the user to navigate back and forward in the session history list. See figure 2.1 This was straight forward and easy to understand for most web users.

The browsers also offer a way to link to a certain area of a web page. The fragment identifier (#) was originally used for this purpose. Any element given an id attribute could be linked to with the fragment identifier. On the src attribute of an anchor element, which usually is used for placing URI's leading to other pages, adding a "#" and the id of the element, the document would scroll leaving the element in question at the top of the window. See listings 2.12.2.

```
<a href="www.example.no/#goToSection">
```

Listing 2.1: The anchor element with fragment identifier (#).

```
https://www.example.com/#goToSection
```

Listing 2.2: The browser's address field after clicking a link with a fragment identifier.

The new History API With the new HTML5 specification the pushState and the "popstate" event simplifies client-side state management. By pushing state objects to the History object with the pushState and replaceState methods the browser can easily save complex state objects and return them whenever the navigation buttons of the browser is clicked. The pushState takes three arguments. The first is the state to be saved, the second an optional name for that state and the third argument is the URI to identify that type of state. When pushing the object "user":"John Doe" and setting the URI to "user" the URI adds the URI argument of the pushState method to the existing URI. See listing 2.3 - 2.5.

```
window.history.pushState({"user":"John Doe"}, null, "user");
```

Listing 2.3: Pushing state with the History API.

```
https://hyperbrowser.uio.no/state/user
```

Listing 2.4: The URI when pushing user:"John Doe" to the History.

```
{user:"John Doe"}
```

Listing 2.5: The object returned from the "popstate" event when navigating back.

2.2.2 The Uniform Resource Identifier

The distinction between a URI and a URL is confusing

The URI scheme is meant to be a way of uniquely identifying representations of resources in a network. The URI is a string of characters, formed in a well defined manner, specifying the rules for obtaining a resource. A URI can be classified as a locator specifying the access mechanism of a resource. When used as a locator it is called a Uniform Resource Locator (URL). It can also be classified as a name representing globally unique resources. When used as a name it is called a Unifrom Recourse Name (URN). Combining the two can also be classified as a valid URI (Berners-Lee, Fielding, and Masinter 1998).

The partitioning of URI space into URL's and URN's have caused some confusion in the web community (*URIs, URLs, and URNs: Clarifications and Recommendations 1.0* 2001). Both URL's and URN's are subsets of URI's. For simplicity one can say that a URL is used to identify the location of a resource in time. The URN is the unique name independent of the location of the resource and its primary purpose is to maintain a persistent

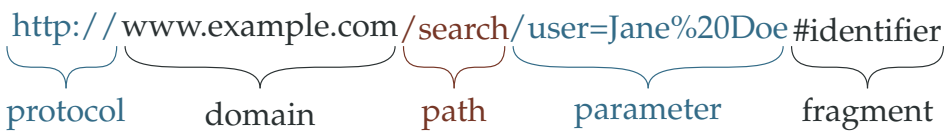


Figure 2.2: The URI annotated.

labeling of a resources also when the resource no longer exists or becomes unavailable.

Both URL's and URN's need to conform to strict rules to be valid URI's. The use case does not define named resources (URN) in any way compliant with the rules of the generic syntax specification for URI's, hence we will not be discussing URN's any further. We will mainly concentrate the discussion on the location of a resource, the URL, but since a URI still identify a resource we will continue using the URI terminology.

The syntactic rules of a URI are overwhelming

The paper defining the syntax of the URI, the "URI Generic Syntax" (RFC2396), encompass 40 pages of detailed descriptions, is only a superset of all the URI schemes available. Examples of URI schemes are ftp, http and mailto. All have their own specifications and are often to much to consider in detail for a web developer. It should be mentioned for the record that even though many of these schemes have the names of protocols that does not imply they are the same as the protocol they are named after. It neither imply that access to the URL's recourse is only possible using that protocol. Oftentimes it requires two protocols to reach a document. When accessing the resource of a http scheme it requires both the HTTP and the DNS protocol (Berners-Lee, Fielding, and Masinter 1998).

Since defining most of the URI schemes of the web is already done and the deciding on naming hierarchies for resources is done by those developing the server, URI grammar is not relevant for the front-end developer. The front-end developer only needs to use the predefined schemes when requesting resources.

However, true this might be on a strict level, this argument might not apply equally well when considering the state of an application. Specifically when utilizing the HTML5 specifications of the History API or the fragment identifier for storing state, the rules of the URI schemes might be of some importance. For instance, the use of the forward slash character "/", the fragment identifier or hash "#", and the colon ":"" are all restricted characters with special meaning when used in a URI. The forward slash divides actors in a hierarchical relationship, the hash has traditionally been used to index certain parts of a document and the colon plays a central part when We discuss a third party library in relation to state management. See figure 2.2 for an explanation on the different parts of the URI.

The fragment identifier

Servers do not interpret the fragment identifier part of the URI string and has therefore also been used as a place to store state specific information used by the client program. Strictly speaking the fragment identifier is not an official part of a URI, but its use in conjunction with the URI and the important role it plays when handling state on a client should be enough to defend the space we have set aside to discussing the issues of the URI.

2.2.3 JavaScript

> JavaScript is ”[...] clearly a grossly substandard language by modern standards and thus can be legitimately described as a toy language.” — (Arno 2010)

JavaScript has had a rapid evolution the last 10 years. The emergence of big JavaScript libraries and frameworks like Backbone and Angular can be seen as a direct consequence of what has been coined the second browser war (Yule and Blustein 2013). Especially after Google launched their Chrome browser in 2008 and the JavaScript runtime Nodejs was build on top of chromes V8 engine, the speed of interpreting JavaScript has increased enormously. Other JavaScript engines now use dynamic compilation, also called "Just in time" technology and the V8 compiles directly to bytecode which make them even faster (Anand and Saxena 2013). JavaScript is also the standard programming language of the "World Wide Web" (Flanagan 2011, p. 1) and is thereby one of the most used programming languages.

Some important features of the JavaScript language necessary to follow the implementation details will be presented in the next sections.

The JavaScript prototype chain

When a JavaScript interpreter starts interpreting a JavaScript program it creates a global object which holds properties common to all other objects. For web-browsers, this is the root window object which was introduced in the web client section 2.2.1. Although most JavaScript programs run in a web browser they do not need a web browser to run. Node is a server-side JavaScript interpreter built on top of the Google V8 engine. Node also creates a global object upon initialization.

All JavaScript objects are equipped with a default property, "this", which refer to the object itself. It also has an "invisible" pointer to a prototype object. When an object is created without specifying a prototype object the prototype pointer, often visible in browsers console and labeled __proto__, automatically points to the prototype object of the global object. This prototype of the global object holds all the functionality common to all JavaScript objects.

When creating new objects, depending on the way they are created, the object can be associated with other prototype objects, which themselves are linked to another prototype object, by the use of the "invisible" __proto__

pointer, all the way to the Object.prototype. This creates a chain of "ancestor" objects.

When a method is called on an object and the object itself does not contain that method, the "this" keyword of the object is "passed" to the first prototype object in the prototype chain. If that object contains the called method, that method is resolved using the "this" keyword of the calling object. If the method is not contained within that object either, the call is being passed down the prototype chain consulting each prototype object all the way down to the global object until it finds a prototype with the called method name.

Call, bind, apply

To understand some of the functionality of the use case there are three important methods, the call, bind, and apply methods of the Function object, that needs to be explained. The Function object is the base constructor functions for all functions associating all functions with the Function.prototype object and thereby all the common function methods. These methods allow you to invoke any function with any object, even when this object does not contain that specific functionality in the prototype chain as if it was a method of the object itself. By passing in a specified object pointer to any of these methods they bind the "this" keyword inside the function with the "this" keyword of the passed in object. See listing 2.6 for a code example.

```
var Person = function(name) {
  this.name = name;
}
var sayName = function() {
  return "My name is " + this.name;
}

var john = new Person("John");

sayName.call(john);

//"My name is John"
```

Listing 2.6: Using Function.prototype.call.

Functions as objects

JavaScript functions are also objects. They can, for instance, contain their own functions, be assigned to variables, and be passed to other functions. This last quality enables the concept of asynchronicity with a "callback" being the passed around function. When, for instance, the jQuery.ajax method makes a request to a server, the calling object pass success and an error callbacks to the ajax method. If and when the server can fulfill the request, it calls the attached success method, if not, it calls the error callback method. This enables the calling program to continue executing

client-side code while waiting for the success or error methods to be called, and thereby create asynchronicity.

```
jQuery.ajax({
          type: 'post',
          url: "a full web address, // can also be a relative
              path to the base path of the website
          data: "data to be sent", // can also be XML or JSON
              data
          beforeSend: function () {
              //if preparation code is necessary
          },
          success: function (data) {
              //Do some action
          },
          error: function (XMLHttpRequest, textStatus,
              errorThrown) {
            //Do some error handling
          }
            });
```

Listing 2.7: The jQuery ajax method

The concept of callbacks is essential to the communications between objects in the implementation of the use case (5.5).

2.2.4 Ajax

Asynchronicity and the jQuery ajax method were introduced in the previous section. The technique of using the XMLHttpRequest object to request data from a server, without the need to get a whole document and thereby enabling asynchronous program flow on the client, has been coined "Asynchronous JavaScript and XML", Ajax for short.

This technique of utilizing ajax to dynamically update a web page has made the revolution of dynamic creation of content in web sites possible.

2.2.5 Software design principles

The art of programming software has been refined for more than a century. Mistakes have been made and knowledge has been gained from the mistakes. With the object-oriented software design tradition, specific principles have emerged and resulted in several general guidelines for good software design. Since these principles, for the most part, apply to every object-oriented programming languages we will not review them here, just give a reminder of some of the most salient points concerning the implementation of the state management application under scrutiny.

Loose coupling

The principle of loose coupling might have come about as a result of programs being difficult to extend or reuse. When one object instantiate another a dependency relationship is created from the first to the other

object. This tight coupling may create problems when new functionality needs to be added to the program.

When object names need to be hard coded to be identified in other objects there is an even tighter coupling of the objects. Such entwined objects should be avoided to promote extensibility and reusability of code.

Classical design patterns

Classical design patterns carry a double meaning. On the one hand "classical" design patterns refer to well established and thoroughly tested coding practices (Gamma et al. 1996, p. 12). On the other hand "classical" design patterns refer to object-oriented programming where classes act as a template for object production (Stefanov 2010, p. 115). The "classical" and highly influential book "Design patterns: Elements of Reusable Object-Oriented Software" suggests as a starting point twenty-three named object-oriented design patterns. The "main goals of the book were to capture and record experience on design in a form that people can use effectively" (Gamma et al. 1996, p. 12). Good design patterns with thereto thoughtful and descriptive names can both establish a vocabulary that can ease communication between experienced programmers, assist in faster comprehension of object-oriented programming principles and function as cognitive aids for junior programmers and programming students.

The observer pattern. The intent of the Observer pattern is to “define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically” (Gamma et al. 1996, p. 9). See figure 2.3. It is often described as a publish-subscriber relationship where the observers subscribe to events and the Subject "publishes" events to the subscribers letting them know there is a new "publication" ready. The "publication" can, for instance, be a state change in the publisher or other related code. Following the publisher-subscriber metaphor a little further; when a newspaper is ready for publication there has usually been two ways for news readers to obtain a copy of the paper, one by going to a newsstand buying it there, the other to subscribe and thereby get it delivered to the front door (Stefanov 2010, p. 171-174). The Observer pattern is implemented by numerous graphical user interfaces (GUI) such as Java Swing and the web browsers event mechanisms.

2.2.6 General web development considerations

One of the challenges of writing nontrivial JavaScript client-side programs is to ensure they run correctly on the wide variety of different browser implementations we have today (Flanagan 2011, p. 325). One example of the discrepancy between browser vendors JavaScript implementation is Microsofts reluctance to implement the DOM Level 2 Events specification which is The World Wide Web Consortium (W3C) recommended standard (*Document Object Model (DOM) Level 2 Events Specification* 2000). This

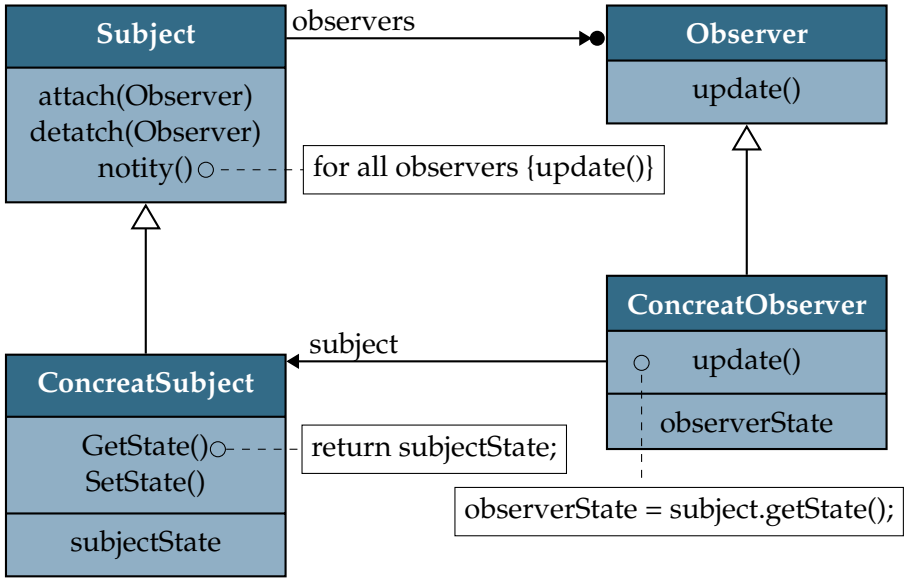


Figure 2.3: The "Classical" Observer pattern as described by Gamma et al.(Gamma et al. 1996).

specification includes crucial functionality such as the AddEventListener which is paramount to most interaction with a web page.

To accommodate for these differences the programmer needs to write a lot of extra boilerplate code if the use of natively supported JavaScript is important. What many chooses to do instead is to use an external third-party library to query the DOM.

JQuery to the rescue

With the plethora of JavaScript libraries available, jQuery is the most widely used according to the Libscore search engine (*jQuery reigns as top JavaScript library* 2014). The ease of use and the great amount of functionality besides querying the DOM, most significantly their handling of Ajax calls has been the leading reasons why we have chosen jQuery over other libraries such as Mootools and Dojo.

The browser implementations of javaScript is not a big problem with modern browsers

Since the release of Internet Explorer 9, however, Microsoft finally decided to implement the decade old W3C standardizations and as a result, most of the browser differences disappeared.

One might think that this would be the end of the widespread use of JavaScript libraries but that has not been the case. A quick google search on jQuery usage statistics showed that between 60 and 70% of the top 10k websites still rely on jQuery (*jQuery* 2016) (*Usage statistics and market share of JQuery for websites* 2016) (*jQuery Usage Statistics* 2016).

Even though most of the implementation differences has disappeared there still exist discrepancies between the browsers as will be discussed in section 6.3.10.

The need to support legacy browsers

A natural explanation for the continuing popularity of jQuery is the need for many websites to provide satisfactory service to legacy browsers such as Microsofts Internet Explorer 8 and before. This also applies when developing new functionality to the use case of this thesis. The use cases goal of being a service available to the public leaves no choice but to cater for the communities where replacing the computer for every third release of new software is unaffordable.

New features still have different implementations in different browsers

The vast amount of mobile and tablet browsers running on different software but also regular browsers for desktop computers does not implement new features consistently or at the same time. New HTML5 technologies such as Server-sent events and the file API are still not implemented by the newest Microsoft layout engine, the EdgeHTML (*Comparison of layout engines HTML5* 2016).

Caching strategies between browsers vary

Another crucial functionality that is implemented differently in a wide variety of browsers is the browsers caching strategies in relation to retrieving bookmarks and back and forward buttons. Do they restore pages from cache or do they reload the page from the server? This problem is not remedied by jQuery or any of the other JavaScript frameworks. Related to the same issue is the difference in implementation of the location assign and location reload functionality. These problems are fundamental to state management and will be brought back to the discussion as it closes in on the difficulties experienced with the concrete implementation of the StateApp.

What is an iframe

The HTML inline frame element or iframe is a special kind of HTML element. It allows for embedding a whole HTML page within this one element. This means the iframe will be a unique window with its own document object model (DOM) and its own unique unified resource identifier (URI). It will have a separate head and body element and all the properties of the window element will be unique and separate from the containing window element. This implicates several challenges for the developer.

Before going into these challenges it is important to remove any misconception regarding the use of iframes. Many people believe that

iframes are bad. This might stem from the fact that the World Wide Web Consortiums (W3C) decided to remove the frame and frameset elements from their recommendations. The reason these items are now obsolete in browsers is that they represented another document model which was not ideal for a lot of reasons. These reasons are the same reasons the iframe should not be used to separate content and will be discussed shortly.

Iframes are useful Iframes are not all bad. The loading of content from different servers can be useful in many different contexts. For instance, embedding video from third-party providers such as youtube or Vimeo, a Google map or a facebook feed in an iframe can enhance a web page. The iframe provides an easy way to create mashups of highly useful content and it is as easy as copy and pasting code. The code needed is often made available by the content providers.

In many cases using an iframe for displaying external content is not necessary. Content providers often offer data in other formats that do not require iframes. Formats like XML and JSON are popular and can be collected with ajax or comet technologies such as the HTML5 API Server-sent events. These technologies give more control of the content for the developer but requires a lot more work.

Another benefit of loading third-party content using iframes is that it enables providers to apply fixes and future updates to the content without the local developer needing to know. This way users of content don't need to worry about compatibility issues when new technologies break current standards.

Allowing third-party programs onto a site could, however, be a security risk. Programs running in an iframe can steal information from the containing site, but because of the strict rules of the Same-Origin policy in browsers and the sandboxing attribute of iframes such risks are minimal. These problems often occur because iframe users have eased the restrictions of the iframe and thereby allowed the third-party scripts to do their attacks.

Iframes can also be used to seamlessly add authentication with OAuth services and it can be used as a means to save state. The latter technique was often used for storing state with legacy browsers before the HTML5 History api. This technique is unfortunately not sufficient to enable bookmarkable state and sharing of URIs and is therefore not elaborated any further.

Gmail is an example of a big site that uses iframes extensively. On a regular Gmail account, Gmail uses 11 iframes, none of which are used to separate content.

Iframes represent separate browsing contexts

The protocol, host and port portions of a URI defines the origin of a document (Flanagan 2011). Since every iframe has its own URI it also has its own window object and its own DOM and it might have a different origin than the containing window. This means that iframes might be

subjects to the "Same-Origin Policy". The Same-Origin policy is a security measure to prevent third-party malicious code to steal information from or manipulate other parts of a program. This is however not an issue with the use case under study so the Same-Origin policy will not be elaborated on any further.

2.2.7 Content separation using iframes

As early as in 1997 did usability guru Jacob Nilsen proclaim that using frames to separate content were the first mistake web designers do. "It breaks the fundamental user model of the web page. All of a sudden, you cannot bookmark the current page and return to it (Jakob Nielsen 1997). Still some websites continue using frames to separate content, now in the form of iframes.

Designing a website using iframes to separate content also breaks with one of the fundamental ideas behind the World Wide Web; every web resource should have a unique identifier such that it can be reached, without ambiguity, from any other document (*Architecture of the World Wide Web, Volume One* 2016). When resources have unique URI's they can form the basis of a stable and lasting web that facilitates the sharing of knowledge and ideas. An iframe represents a separate resource from its containing window. It has a separate URI possibly with its own separate state encoded on its fragment identifier. An accurate representation of the containing windows state must be merged from all the states of the iframes it contains to be reached unambiguously. There exists no such merging mechanism by default in browsers.

A scenario pictured from the world of bioinformatics might give a clearer understanding of what can be lost when a website does not handle frame state: A researcher creates a hypothesis using a web tool from a project such as the Galaxy platform. She wants to share her hypothesis in an online article and copies the URI from the Galaxy sites address field and pastes the URI into her article. When later researchers want to replicate her findings they follow the link from the article but the link leads them to the welcome page of the site. The hypothesis is not sharable through the URI.

The state of the hypothesis is not sharable through a link because the Galaxy platform does not collect state in a manner suited to reproduce that state. The state of the iframe in which the hypothesis reside is encoded in that iframes location object—if at all collected—and is never transferred to the address field of the browser. It is only the browsers address bar that is available to the user. The browser has no means to recreate the state from a URI copied from the address field if there is no state encoded in it.

This way of using several iframes to separate content mimics the obsolete HTML frameset element. The frameset element allowed for frames to divide content into sections thereby enabling scrolling and loading of content in one frame without affecting the other sections. This was a welcomed feature for web designers before the advent of Ajax technologies. Now the same effect is better handled with CSS and Ajax.

At the same time state still needs to be collected, stored, and saved somewhere when using ajax as a means to collect new data for a dynamic page. As is the case with iframes saving ajax collected state is not an automated feature of the browsers either. To be able to share information that information somehow needs to be encoded into a unit that is shareable. For the web, this shareable unit has been the URI. By adding frame state to the fragment identifier the uniqueness of the page as an atomic unit of information is preserved. According to Jacob Nielsen, this was part of Tim Bernes-Lee's genius idea for the World Wide Web (Jacob Nielsen 1996).

Not possible to print pages that use iframes for content separation

The last point to this section is that the print functionality of the browser has trouble printing pages that are made up of iframes. Inspecting a printout of the HyperBrowser in figure 2.4 shows that the browser print each frame on top of each other leaving the page unintelligible. This is an obvious drawback of using iframes to separate content but can easily be remedied by providing an extra styles file, print.css, to the index.html file.

Loading

A minor challenge when dealing with iframes is the fact that the elements of the iframe DOM are not ready until the "ready" event is emitted on the window object of the iframe. This means that any listeners must wait for the "ready" event before they can be attached to the elements. This also relates to functionality present on an iframes document, hence it is not possible to depend fully on functionality being present all the time.

For a developer not familiar with browsers loading issues this kind of thinking will be challenging. On the other side, with a little bit of planning and confidence in the event model for communication between objects, one can label these kinds of challenges as trivial.

2.3 Scientific foundations

In this section, we present relevant research and elaborate on the research put forward in the introduction. To help us explore this groundwork we assert that:

1. there is a need to manage state,
2. there are suggested locations for communicating state, but
3. there is no precedence for how to manage state

in a web environment. These assertions will guide the establishment of a foundation, relevance, and scientific value to the thesis discussion.

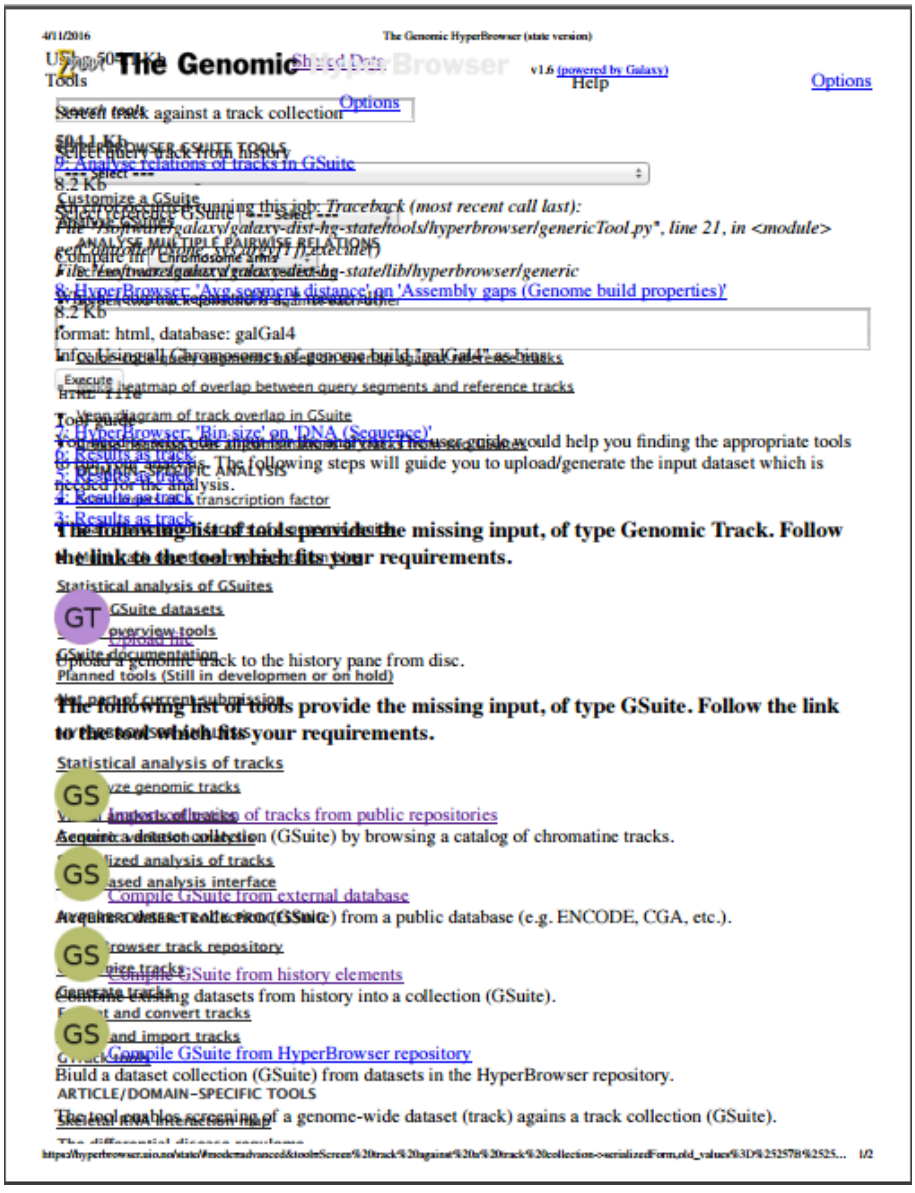


Figure 2.4: An example of a printout from a browser using iframes to separate content.

2.3.1 There exists a need to manage state

Why is state important?

During the last 20 years, research has shown that the use of default browser behavior, particularly the use of the browsers back button, has diminished.

Cockburn et al. suggest changing the way the back button is implemented to remedy web-users misconceptions of the default stack-based behavior (Cockburn and Jones 1996). They propose a temporal back to avoid pruning of the navigation stack after loading a new page. The temporal scheme allows for saving every session history entry and offer two different histories, one stack based and one temporal (Cockburn, McKenzie, and Jasonsmith 2002).

Later studies indicate that the decline in back button use can be explained by a change in browsing behavior to a tab based session history management where the tabs function as a readily available session history list (Obendorf et al. 2007) (Zhang and Zhao 2011). These later studies try to discern different goals of using the back button and tabbed browsing which is not very relevant for the current study.

Nonetheless, these studies also indicate reasons for the change in browsing behavior. Obendorf, for instance, suggests that this trend can be triggered by the lack of state management (Obendorf et al. 2007). They claim that the reluctance to use the back button can stem from the fact that application like websites tend to extend the use of form submission to provide dynamic content, that state is usually lost when going back and that such pages can not be bookmarked.

These assertions can be evaluated as symptoms of negligence and that there is a need to remedy the shortcomings. This thesis is an effort in that direction.

2.3.2 The fragment identifier as a suggested location for storing state

The advent of web services as an effect of the turn towards a dynamic web has sparked scientific interest in the field of state management. Pat Helland distinguishes between data on the inside and data on the outside of a service application. He asserts that the data inside should be encapsulated and only be shared by other services by the means of immutable messages (Helland 2005).

Foster et al. evaluate four different addressing mechanisms for service messaging, two of which are relevant to this discussion. HTTP identify state representations through a URI whereas the "no conventions" approach state that "state representations are not fundamental building blocks" and that "resources should be identified through URIs (or URNs) inside messages, leaving it up to the application domain specific protocols to deal with state management" (Foster et al. 2008).

As logic is moved from the server to the client these client-side applications can resemble web services. When clicking a link in an outside website the action of displaying a certain mashup of external content

possibly gathered from various locations is provided or serviced by the addressed application. The URI then become the message and the address mechanism. The only problem is that the browser always interprets the URI as a request back to the server so the actual state of the link must be encoded by other means. This new way of requesting dynamic diverse information expresses a need for an extra layer of identification.

Since a URI is used to identify a collection of resources for a certain website, i.e. all the documents associated with a certain server, the fragment identifier could be a natural place to associate state specific information. The server dos not interpret that part of the URI so the client-side developer is free to use it as she may see fit.

Kannan et al. propose to use the fragment identifier as a place to put page coordinates. This enables browsers to focus on certain parts of a static display of information, thus making the Internet more dynamic (Kannan and Hussain 2006). Others have used the fragment identifier as a means to apply logical names for specific parts of a document allowing for automatic generation of web pages (Aimar et al. 1995). This can, for instance, be used by semantic technologies.

Windows live uses what has been coined "Fragment Identifier Messaging" to offer mashups of content. Fragment Identifier Messaging uses the fragment identifier part of the URI as a location for encoding the messages passed between the internal frames of an application. Although this use of the fragment identifier disables the fragment identifier as a storage for the client-side state the HTML5 API postMessage mitigate these problems according to Barth et al. (Barth, Jackson, and Mitchell 2009).

2.3.3 The fragment vs the model

Using the fragment identifier as a place to store state has also been proposed by Mikowski et al. (Michael Mikowski and Powell 2014). They present what they call the "Anchor interface pattern" where the anchor, we have called it the label fragment identifier, is the driving force behind all state changes. When any state change occurs in a web application the receiving code for the "change" event should set the URI directly and then promptly return (Michael Mikowski and Powell 2014, p.85-88). Parsing the URI can then be done by code listening for the "hashchange" event of the window object. When other parts of the application need to synchronize they must consult the URI.

The benefit of using the Anchor interface pattern is that it facilitates correct behavior of the default browser functionality.

In connection with their book "Single page web applications", they have developed a jQuery library, the URI anchor, as a utility library for realizing the Anchor interface pattern. The library authors state that the library will help you: "Make your application bookmarks, browser history, the back button, and the forward button act just as the user expects while enabling you to update only the part of the page that has changed." (Mike Mikowski 2016). The functionality of the uriAnchor is described in section (5.3.1).

The recommendation of the Anchor interface pattern stands in opposition to the somewhat more recognized concept of a model as the central unit for synchronization. This concept will be presented in the next few sections.

2.3.4 The View and the Model keep business logic separate from the presentation

> "Because little significant software in a commercial environment is developed using only programming skills, students without exposure to design patterns and frameworks will be ill-prepared for the workforce" (Chao, Parker, and Davey 2013)

The moving of responsibility for handling application logic and data to the client looses the natural separation of presentation in a thin-client web page (view), the application logic on the server (controller) and data in a database (model). When the traditional Model, View, Controller (MVC) separation is lost the DOM will contain an entangled clutter of presentation elements, application logic, and non-visual application data. According to Castro et al. "[...] browser[s] do not have the right programming abstractions to support the MVC approach". They suggest creating an XML representation of the client application state separate from the DOM with an active binding between presentation elements in the DOM and the underlying data DOM. Changes to the data will update the presentation elements by means of emitted events (Castro et al. 2006).

Later JavaScript frameworks like Angular.js takes this binding idea further by introducing two-way binding. In addition to the binding presented by Castro et al., this type of binding also update the client model when a bound view is altered. This can, for instance, happen when a form is changed (*Angularjs* 2016). The Angular two-way binding scheme is however not possible to transfer to situations where altering the HTML generated by an existing server is not feasible.

Backbone, on the other hand, is more like a library that offers utility objects for keeping the business login separate from the user interface (*Backbone.js* 2016). For instance, they provide a View object to handle user input and a model to contain the application state. All the objects of Backbone are extended with an event emitter such that binding views and corresponding models are made easy.

The MVC controversy

Many of the emerging JavaScript frameworks and libraries provide Model View Controller (MVC) like structures where the separation of presentation and data is a prominent feature but where the interpretation of the role of the interacting objects diverge.

Syromiatnikov et al. classify three main families of the MVC architecture and they coin the architecture MV* to indicate the controversy of the

communication between user interface and application state and the role of the controller (Syromiatnikov and Weyns 2014).

The original family of MVC patterns promotes a synchronization scheme where the controller handles user input and relays to the model any user interaction relevant to the models. The model is often oblivious to any other object. When changed they emit "change" events for the listening views to pick up and render the changed model viewable to the user. There are also versions to this MVC family in how the objects communicate. Is it done directly where one object holds a pointer to the other or by notification where objects emit change events picked up by listening objects?

The Model View Presenter, on the other hand, has another understanding of the role of the controller and has given it the name Presenter. The presenter is a mediator of communication between the view and the object. Here the view handles all user interaction. It is responsible for listening to user input as well as displaying model changes. Also with the MVP family, the means of communicating change vary to some extent.

The last family of MV* patterns classified by Syromiatnikov et al. is the MVVM architecture. The MVVM family maintains the original separation of responsibilities between View and Controller but none of these communicate directly with the model. Instead, they introduce a fourth family member, the ViewModel, which handles logic and relays all the information from to and from the model.

All these families can be used both on a server and on a client. Other interpretations of the MVC pattern divide the participating objects between the client and the server. Leff et al. describe the Dual-MVC pattern where the Controller is divided between the client and the server and the fractal MVC describes the MVC architecture as repeated in many of the different layers of a software program.

2.4 Wrapping up the background

The background chapter has introduced the web browser in addition to relevant design principles, patterns, and architectures. Some research concerning how the use of the default browser behaviors has changed over the last 20 years has also presented. A suggestion that this change in behavior might be caused by the lack of support for the mentioned behaviors has also been put forward. We have introduced the dispute of where to store client-side state, on the URI or in a model? This question will be discussed in section 6.3. Lastly, the controversy of how the model should interact with the other participants of the MVC architecture where described. This will be treated in section 6.3.1.

Chapter 3

The use case

3.1 A real world use case

This chapter describes the Genomic HyperBrowser, from now on the HyperBrowser. This website will function as the use case for our investigations of client-side state management. The HyperBrowser is build on top of the Galaxy project which provides most parts of the graphical user interface (GUI) functionality for the HyperBrowser. Nevertheless, we will continue referring to the GUI as it was the HyperBrowser. This is done to simplify the presentation and to avoid confusion.

The HyperBrowser is a well suited use case seen from three distinct perspectives. First, it perfectly demonstrates a website where sharing state can be of general importance. The HyperBrowser HTML form selections represents scientific hypotheses which, if made easier to share, can be a facilitator for scientific reproducibility.

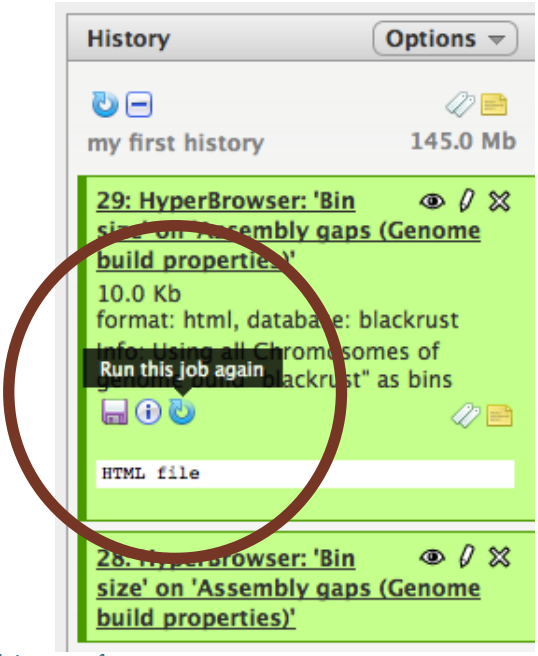
Second, the HyperBrowser does not aggregate state in a manner which makes the state linkable or conform to the default browser functionality.

Third, the way the HyperBrowser GUI is designed poses interesting challenges worth investigating in relation to state management. This is especially due to the use of iframes as a means to visually separate content on screen.

It should be mentioned that there already is a way to share hypotheses using the HyperBrowser interface but this involves a few steps and the need of guidance. Sharing the state of tools in the URI will enable linking directly to the hypotheses from other web recourses.

3.1.1 The Genomic HyperBrowser

The intention of the Internet and the World-wide web to augment the human intellect, from the ideas of Vannevar Bush and Doug Engelbart to the realization by Sir Tim Bernes-Lee, has been motivating many a web project (Rheingold 1985) (Berners-Lee, Cailliau, et al. 2010). This is also the grounds on which the HyperBrowsers was born, to help researchers do better research on genomic data. The HyperBrowser provides tools to aid researchers in defining advanced statistical hypotheses on genomic data.



The history frame

Figure 3.1: To recreate the hypothesis the "Run this job again icon" must be clicked.

The fundamental idea of the Internet and the World-wide web is that of sharing information and knowledge by linking between documents. The HyperBrowser facilitates sharing of results and also makes possible the sharing of hypotheses but this sharing is tedious and not available to linking. To share the state of a hypothesis the recipient need to be a registered HyperBrowser user. The next step is specify to share a particular HyperBrowser history with the the recipient. Then the history item that contains the results from the analyzed question needs to be outlined verbally or in writing and the recipient needs to press the "Run this job again" icon as depicted in figure 3.1.

The HyperBrowser project was built to enable processing and analysis of genome-scale datasets (Sandve et al. 2013). A single dataset can be analyzed against a reference genome or a pair of datasets can be analyzed against each other or against a reference genome.

3.1.2 The GSuite HyperBrowser - A need to simplify the GUI

The GSuite HyperBrowser project, from now on the GSuite, is an extension of the HyperBrowser and facilitates analyzes done on collections of datasets (Simovski et al. n.d.). This increase the amount of available choices and could be perceived as intimidating to unexperienced users. When unfamiliar with the user interface or with making complex statistical hypothesis based on genome data, the users might be overloaded with the amount of choices and of the information presented by all the new possible

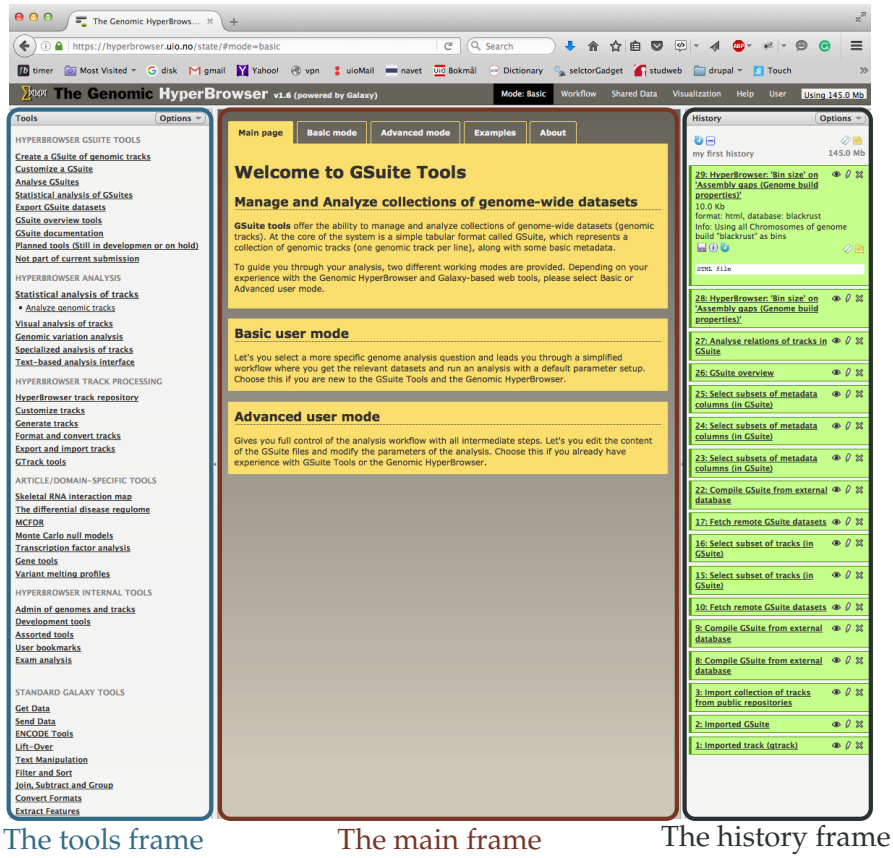


Figure 3.2: The welcome page of the GSuite HyperBrowser Graphical User Interface (GUI).

choices of the GSuite. The hiding and revealing of information in a basic or advanced mode is the second functionality the StateApp will provide for the HyperBrowser and the new GSuite project.

3.2 Describing the HyperBrowser (Galaxy) GUI

The HyperBrowser GUI consists of four distinct parts. The logo and main navigation bar is located on top of the browser window. The rest of the page is split in three vertically aligned sections. See figure 3.2. The left most section consists of a list of links to genome specific tools. The next section, the middle one, is the main section of the HyperBrowser. The right-most section consists of history elements. This thesis does not interfere with the history section and will not be discussing this any further. Clicking on one of the tools in the tool links section opens the corresponding tool in the main section.

A tool consists of a HTML form with several options to choose from and a button to send the form to the server. This button is often called execute or start analysis. When a tool is executed a confirmation notice is displayed in the main section of the site and a history element is added to the history

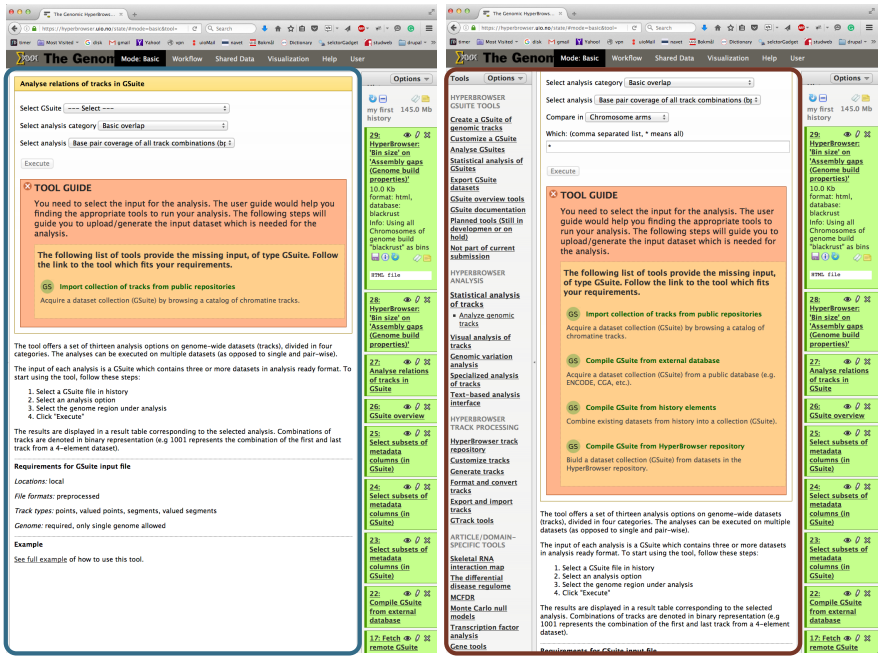


Figure 3.3: The welcome page of the GSuite HyperBrowser Graphical User Interface (GUI)

section.

Some tools have a dual appearance. These tools are part of the GSuite project. They have both a basic and an advanced representation. Only one of these appearances are sent from the server depending on a checkbox present within the tool form. See figure 3.3 for details on the two GSuite tool representations.

The main section also displays the welcome page when opening the HyperBrowser. The welcome page is one of several tabs in the main section. The two other tabs important to this thesis is the basic and the advanced tab. These tabs display guides to how to use the tools provided by the GSuite. The basic tab leads the user through a simplified workflow while the advanced tab gives the user full control of the analysis workflow. In the rest of this thesis we will be calling the above described tabs the GSuite guides. Important to mention is that these tabs are not present when displaying a tool in the main frame.

3.3 Requirement specifications for the HyperBrowser StateApp

This section describes the specifications for the application made in conjunction with the investigations of client-side state management.

1. A basic / advanced mode should be able to be set from one button placed in the main navigation, from a small triangle on the border

between the tools and the main iframe, with tabs within the main iframe when available and from within certain GSuite tools. When setting mode in one place, the mode should be updated all the other relevant places.

2. Implement the browser functionalities of the back and forward buttons, the bookmark functionality and make the HyperBrowser tool state sharable.

The first specification concerns hiding information. The HyperBrowser offers a vast amount of tools and every tool provides a lot of possible choices. From psychological experiments to web usability gurus such as Steve Kruge and Jakob Nielsen we have learned that too many choices makes us unhappy (Schwartz 2004) (Krug 2005) (Jakob Nielsen 1997). This well known web design / UX principle states that unexperienced users easily get confused when there is too much information to take in at one time.

The proposed solution to help new and unexperienced users is to present two "versions" of the site, a basic and an advanced mode. The advanced version will display the page as it is today, that is it will stay unchanged. The basic user mode will conceal the tool list and only present a guide to some basic tools. The guide is contained within the GSuite and will not be a part of the specification for this thesis. The GSuite also offer tools with the basic/ advanced distinction where tools presented in basic mode will have less choices than those in advanced mode. Relevant to this thesis is the integration of the GSuite basic / advanced mode to the user interface of the Genomic HyperBrowser.

The second specification will deal with the fact that users expect certain behavior from a website (Michael Mikowski and Powell 2014) (Jakob Nielsen 1997). The current implementation of the HyperBrowser does not adhere to the expected feature of bookmarking the state.

The ability to save state will enable the researcher to make their work more accessible to the public which will promote reproducibility. By letting researchers easily share their research questions and hypotheses online, in articles or other documents, the transparency of research can be increased.

3.4 A definition of HyperBrowser state

State, in the context of the HyperBrowser in combination with the StateApp, is defined as the current application mode, either basic or advanced, and the current serialization of the tool form whenever such a form is present on screen.

Chapter 4

The process of acquiring knowledge

Embarking on the journey of investigating state in a web environment has had its challenges. This chapter is devoted to recounting this journey. It will not be a thorough review of the tools and frameworks used but rather a tale of all the work that has been laid down and all the learning accomplished during development. This has been a personal endeavor with little to no technical supervision. I will, therefore, only for this chapter, put aside the notion of a "we" and instead use the first person "I".

The inclination for client-side state management and the special interest of applying state management on an existing website with a somewhat complex state logic led me to pursuit a well-fit use case rather than personified technical supervision. Even though knowing that this would lead to a steeper learning curve the benefit of having a supervisor familiar with the use case was far superior to that of a supervisor fluent in the technicalities of web development. Technical guidance is easy to obtain from the World-wide web which is not the situation for supervision on a specific use case.

The account of this journey will pass through three stages: The planning stage, the implementation stage, and the refactoring stage.

4.1 The planning stage

Planning a web project involves decisions concerning things like how to organize the different parts of the JavaScript program, how to include third-party software and how to load them all onto the website.

To take the last point first: Solutions such as letting all custom code reside in one gigantic file seems like a good idea when it comes to performance. The browser only has to make one request to the server to get all the code it needs. On the other hand developing in one big file will be tedious. Vice versa, dividing all the different parts of the program into small units of code and then let every single part have its own script tag would create pointless requests to the server. The server would get an unnecessary workload and it would add to the ever increasing traffic of

our common network resources. Adding an extra script tag for each tiny third-party helper program would also lead to unwanted pollution of the global namespace.

4.1.1 Building and package managing tools

For server-side JavaScript development Node (*nodejs* 2016) and the Node Package Manager (NPM) (*npm* 2016) provides a solution to integrate third-party software into the code base of a program. NPM handles the downloading and organizing of external software and Node the "requiring" of the software into the program under development. The developer only needs to add one line at the top of a file to request the desired plugin and the functionality will be readily available. See figure 4.1.1.

Until browserify came along there was no such way to integrate all the third-party frameworks, libraries and plugins without adding a separate script tag in the HTML document for each one. Browserify brings the "require" method of Node to the browser while utilizing npm for package management. This is, however, not the only benefit of using browserify. Browserify also enables requiring of custom code. This means that the developer can divide code into small readable modules and "require" the code whenever the code is needed. Browserify then merges all the files into one. The benefit is a better encapsulation of functionality and preventing the pollution of the global namespace. To summarize, browserify merges both the custom made code and the external software into one big file such that this one file can be added to a script tag on the main HTML page of the website.

```
var _ = require('underscore');
_.extend(ModeModel, function([...]) {};
```

Listing 4.1: Example of the browserify require statement.

Gulp is another utility program that helps development by automating the build process. Every time the developer needs to test a small code change she needs to let browserify merge all the files before sending the one file to the browser with the new changes. The browserify compiling is not an unaffordable task to do manually. It is when adding tasks like testing and other time-saving developer tricks that build tools like gulp can save the developers a lot of time.

Watching your JavaScript code for syntax errors, compiling .sass files to .css, uglifying and enabling differentiation between development and production files are just a small portion of what a junior developer needs to understand and consider when starting a web development project. The downside is the learning curve for how to utilize all these tools and how to set them up with Gulp. This is not trivial but doable when spending the time necessary to understand the basics.

The next sections will be devoted to introducing Behavior Driven Development (BDD) using Jasmine. For people coming from test-driven

development, learning behavior-driven development is easy. With little or no prior testing knowledge the challenge can be daunting.

4.1.2 Behavior Driven Development

The decision to adapt to an agile software development process was made not only from a desire to learn new tools and methodologies, nor as just a means to develop faster and more robust software but also came out of pure necessity. The use case resides on a production server where scientists do their daily calculations and scientific endeavors. These calculations could be requiring enormous amounts of processor resources which, occasionally, resulted in slow servers. Uploading code could take more than 60 seconds. That is unacceptable when developing code. Small changes need to be tested all the time.

Downloading the whole code base of the use case was not a solution, either, since the project was gigantic. A better solution was to do some of the testings locally. This could have been done manually with any web browser open and the technique of saving, building and refreshing. Nevertheless, writing tests has the benefit of being runnable every time a new feature is added and potentially can break the code base. Using tests as a means to plan and understand the functionality before implementing it may also be an advantage. Since the build tool Gulp already makes it fairly easy to add testing to the build process the decision to learn Test Driven Development (TDD) was easy.

Behavior Driven Development (BDD) was the new trend in the JavaScript and Node testing communities and sounded like a good way to go about driving the development of the program. BDD is mostly like TDD but propose to use natural language to describe the testing scenarios. This turn toward a more descriptive way to present the testing was partly a consequence of developers frequent problems to find good starting points for the tests or to determine which aspects of the program needed testing and which did not. It was also a means to facilitate better communication between developers and stakeholders (Soeken, Wille, and Drechsler 2012).

Jasmine

The Jasmine framework was chosen because it is an all in one solution. It integrates both an expectation library and a mocking and stubbing library within the boundaries of the testing framework. The other popular solution is to use the testing framework Mocca with the expectation library Chai and use Sinon when needing a mocking and stubbing library. They are very much alike and choosing one over the other would probably not mean a lot.

An example of a Jasmine test is shown in the code example 4.2 and the accompanied excerpt from the terminal printout of the Jasmin test can be seen in figure 4.1.

```
describe("A modeView Prototype ", function() {
  it("is defined", function() {
      expect(ModeView).not.toBeUndefined();
  });

  it("provides the methods of the BASE VIEW prototype object"
      , function() {
      expect( _.isFunction(ModeView.initialize) ).toBe(true);
      expect( _.isFunction(ModeView.render) ).toBe(true);
      expect( _.isFunction(ModeView.setElement) ).toBe(true);
      modeViewInst = null;
  });
```

Listing 4.2: Example of behavior-driven development using Jasmine.

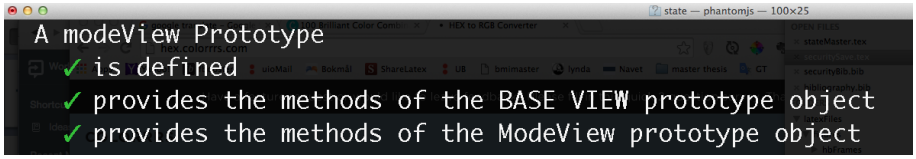


Figure 4.1: An excerpt of a Jasmin terminal printout.

Integration testing with Jasmine can be quite tricky

Even though Jasmin is mostly used as a tool for testing units of code, it seemed like a good idea to test the interaction of the objects as well. There were so many objects that needed to interact with others to be useful, so I decided to use Jasmin also as an integration testing tool. I also found a plugin, the jasmine-jquery plugin, that enabled testing jQuery code and stub the DOM. This meant that there would be less time spent doing manual testing on the actual live website.

In the beginning of development, this worked out smoothly but as the code base grew larger testing the interaction of objects became more and more time consuming. It was hard to decide if the test units actually tested what they should and if passed tests really were passing because they worked. Did it test the right code? I also discovered cases where the tests failed but should not have. After about 100 working test cases I stopped the behavior-driven development phase and began the tedious activity of manual testing on the web.

End to end testing was too complex to add to the workflow. After looking into some end to end testing frameworks I decided it was unreasonably complex to anticipate what to expect from a returning test. Even though I knew that testing manually on the web would be very time-consuming, the alternative seemed too hard to get into without any external help. The problem was also that the understanding of what was actually under development was difficult to grasp at the early stages of development.

4.2 The implementation stage

We have already accounted for some of the early challenges of the implementation stage. Learning the software and make them work together smoothly took a lot of time but resulted in a more focused implementation.

4.2.1 Emulating Backbone for data binding

The aspiration to understand the inner workings of a JavaScript framework was one of the decisive reasons behind emulating the Backbone library for the binding functionality: the binding between business data and presentation. Another popular JavaScript framework, Angular, had been tested in a preliminary study prior to the current project but were found useless for extending an existing program. The main point of Angular is to provide a two-way binding scheme where the developer adds directives directly to the HTML document. Since the use case already provides most of the HTML, Angular would not add anything to the project except for complexity and a big code base.

Object creation scheme

Backbone was also an interesting library to study because of the choice of object creation scheme. They have chosen to emulate the way "classical" programming languages create objects, with constructors and the new operator. I wanted to go the other way and try to understand the powers of using the language as it was designed, that is as a real object-oriented language without the notion of a class template. This way I was assured that I could never just copy the code but had to understand every tiny little bit of the code I wrote. This was very rewarding and gave me a great foundation for the challenges to come during the refactoring stage. There is, however, no space nor the time to give an account of all the lessons learned during this exploration. It suffices to say that my understanding of the JavaScript language was augmented.

4.2.2 Development on a living server - Insilico

Developing on the Insilico server could sometimes be quite challenging. As mentioned earlier some days committing a change to the server could take up to 60 seconds. This is not ideal. If using the server only as version control and thereby only committing every time a feature is tested and ready this might not be a big problem. When needing the committed code for testing small features every five or ten minutes waiting this long amounts to a lot of waiting over the course of a months development. Needless to say, this became an annoyance. Especially until I understood that some of the time waited were because of the commit script I got from the technical staff of the HyperBrowser. The original commit script searched through the whole code base for updated code. When telling it to

start the search at the folder where the actual code resided, some seconds were saved.

Subversion

The version controlling system used by the HyperBrowser development team is Subversion. Coming from Git I needed to adjust some habits and get used to reading some extra documentation but all in all this was not a great hindrance for development.

4.3 The refactoring stage

Finally, the coding came to an end and it was time to refactor the code. The code was cleaned and some functionality were moved to other objects. A major milestone was reached. The StateApp was ready, time to start writing the thesis.

4.3.1 New specifications

At the time of the implementation I developed the mode and state management functionality without being aware that the GSuite team wanted the functionality the StateApp provided. When presenting the finished product I was asked to integrate the application with the GSuite project.

Integrating the application with the GSuite project became a great learning experience. Six full-time employees, post doctors and research fellows got to test the StateApp in their daily working environment. New functionalities were requested and bugs were continually found and fixed.

4.3.2 The enlightenments of bugs

Notwithstanding that during the next couple of months, while refining the functionality of the StateApp, I got valuable insight into the process of developing a program in a team, none of the six other developers found or at least mentioned the most severe deficiency of the StateApp: that it had corrupted the default browser back button behavior. This was a behavior that was working before I started developing the StateApp and the fact that it was broken was not found until months after the refactoring stage was officially ended.

This new period of searching, testing and trying to comprehend the reason behind this major malfunction gave me a lot of new knowledge and turned out to become a valuable asset, as it became an important part of the discussion for this thesis.

4.3.3 The acknowledgment of a job well done

As the state management functionality of the StateApp has become an integral part of the GSuite project it has gotten some good feedback and

recognition. It has been appreciated for enhancing the workflow and making recoveries a lot easier.

The final statement of recognition is the fact that the StateApp will be part of a research endeavor about to be submitted for publication.

Chapter 5

Implementation

To investigate the challenges of applying state management on an existing, server centered, web application we have developed what we have called "the StateApp". The result is a working JavaScript program that adds state management to a website where such state did not exist originally. The intention of this chapter is to add details relevant for the discussion and not be a thorough review of the whole code base. The implementation described in this chapter is far from flawless but will serve as a starting point for discussing design principles and good coding practices. The reasoning for the choices made will also be explained in the discussion.

5.1 The functionality

The StateApp has two main tasks. The first is to make sure that all parts of the HyperBrowser are synchronized with a "modeState". There are two modes: a basic and an advanced mode. The second is to record the state of the HyperBrowser, the "toolState".

1. ModeState: The mode state is effecting three parts of the page.
   (a) a) the left sidebar where the tool section is located. This should be closed when in basic mode and open when in advanced mode.
   (b) b) the GSuite tools have two different representations delivered from the server. The StateApp needs to know if the tool returned from the server is in basic or advanced mode and synchronize the rest of the application accordingly. Also, when a mode change is triggered from anywhere in the application the corresponding tool representations should be fetched from the server.
   (c) c) the GSuite guides should be synchronized with the current page mode. When the mode is changed somewhere outside the GSuite guides the StateApp should display the tab corresponding to the mode. It should also toggle the mode on the rest of the application when a user switches from one tab to the other.

2. ToolState: The StateApp should record the state of the HyperBrowser to enable the default behaviors of web browsers to work as expected. By storing state on the fragment identifier of the URI, the StateApp can restore the application state. This enables the correct behavior of the back and forward buttons. It also enables the user to save the state as a regular bookmark or send the URL of a specific tool selection to a colleague or other interested party. ToolState is however only available for 24 hours as is a specification of the HyperBrowser. After this time limit, the toolState is erased from memory.

5.2 The program execution

As is the case for all JavaScript programs run in a web browser environment the execution of the code has two distinct phases. The initialization phase and the event phase (Flanagan 2011, p.317). In addition to the two phases of browser execution, the StateApp also goes through its own initialization phase before it enters the event phase.

5.2.1 The initialization phase

The initialization phase is triggered by the "ready" event emitted by the root windows DOM and starts out by creating a uriParser object. Before starting the uriParser object the StateApp creates what we have called the "modeApp". The tasks of the modeApp object are to create a modeModel, a modeView, and a modeController. It then sets up mode related event listeners and returns a pointer to the modeModel as this is needed by other objects. The "modeApp" is also responsible for appending the "basic/ advanced" button onto the DOM and more specifically to the main navigation bar.

After the modeApp has returned, the toolApp is started. The toolApp is injected with the modeModel and creates a toolModel, a toolView, and a toolController. As with the modeApp, the toolApp now sets up event listeners related to recording the state of the tools. Since the tools are basically HTML forms, the toolApp is set up to serialize the form using the jQuery serialize method. The form serialization is added to the toolModel. The form is serialized on every reload of the main frame. When the event listeners are all set up the program returns to the main initialization code. The toolApp has been assigned with other responsibilities as well. These are not part of the initialization routine and will be described in section 5.3.3.

On returning from the toolApp, the uriParser object is started. It first checks to see if the hash part of the windows location property is set. Unless the location hash is set the program continues onto consult the browser's localStorage. If no state is stored on the localStorage the program sets itself with the basic mode. If localStorage has mode information, the uriParser object triggers a history:change event. Attached to the event is the state object stored on localStorage. This is also the case if the location hash is set:

the program triggers a history:change event with the parsed URI object attached instead of the state object from localStorage. The initialization process is now terminated and marks the start of the second phase of the program, the event phase.

5.2.2 The event phase

After the initialization has terminated the app enters the event phase. All the objects created in the initialization process has been extended with a common dispatcher object allowing internal communication within the objects of the app. This internal communication is not in any way connected with the browsers event mechanism even though they are similar.

The dispatcher object of the StateApp is designed as an Subject and an observer in one object. By extending all the other objects with the dispatcher object, every object is made into observers observing the common dispatcher prototype object. See section 2.2.5 for an explanation of the Observer pattern.

When an object needs to listen for a specific event it needs to register a "callback" function to the dispatcher object. When the appropriate event is triggered the listening object gets "notified" and then asks the Subject of its current state. See section 5.5 for details on the dispatcher object. The event phase continues until the browser window is shut down.

In the event phase, the program will have long periods of silence when no code is run. These long stretches of no activity will be interrupted by bursts of activity when a user interacts with the site or the browsers.

5.3 Storing and retrieving state from the URI

To store state on the URI the state must be transformed to a string. The third-party library uriAnchor provides utility methods for parsing and stringifying the part of the URI we will refer to as the fragment identifier. See section 2.2.2 for a detailed discussion on the URI and the fragment identifier.

5.3.1 The uriAnchor library

The main functionality of the uriAnchor library is to stringify complex objects onto the fragment identifier part of the location string and vice versa to parse the location string back to a javaScript object. For instances, in relation to storing the mode state in the StateApp, the mode state can be put in an object and given to the setAnchor method of the uriAnchor library:

```
uriAnchor.setAnchor({mode: 'basic'});
```

Listing 5.1: Setting the location with the Uri Anchor library.

The uriAnchor will set the location.hash for you. This will be reflected in the address field of the browser like so:

```
"https://HyperBrowser.uio.no/state/#mode=basic"
```

Listing 5.2: The browsers address field after using the setAnchor method of the uriAnchor library.

If other state types are needed to be recorded as well, the key/value pair will be separated by an ampersand (&):

```
#mode=basic&otherState=testState
```

Listing 5.3: Standard key/value separation of the URI parameters.

The library also supports storing state objects within objects. It does this by associating an independent property of the state object with a dependent property. The dependent property holds the additional associated state for the independent state property:

```
$.uriAnchor.setAnchor({
mode   : 'advanced',
tool   : 'Analyze genomic tracks',
_tool : {
 serializedForm    : 'dbkey=blackrust',
 currentSelection : 'galaxy_main'
},
otherFutureProperties : 'red'
});
```

Listing 5.4: The uriAnchor library. Parsed complex object.

In listing 4.4 the "mode" and "tool" properties are what has been called, named independent properties. They are named in that they have strings as values and they do not depend on any other properties. The "_tool" property, on the other hand, does not have an independent string value. The value is an object and the properties of that object are associated with the "tool" object through convention. The "_tool" property has been called the dependent variable as it represents additional properties of the associated independent property.

```
#mode=advanced&tool=Analyze%20genomic%20tracks:serializedForm,
    dbkey%3Dblackrust|currentSelection,galaxy_main&
    otherFutureProperties=red
```

Listing 5.5: Example of a stringified complex object.

5.3.2 The uriParser object

The uriParser object acts as the glue between the application state and the browsers address bar. Technically the interface of the browser's address field is the location object of the root window.

Emitting discrete events for the different model states encoded in the URI is done by the uriParser objects setModelState method. Changing the URI triggers the browser's "hashchange" event. This event is caught by the uriParser object. One of the many tasks of the uriParser object is to trigger separate events for each state that is changed. The uriAnchor library simplifies this process of separation. As the uriParser loops through

the state object returned from the uriAnchors makeAnchorMap method, it triggers an event specific to each independent property. If there is a dependent object associated with the property at hand the uriParser object passes this object along with the event.

Making sure that only changed state trigger events are for the time being not enabled. This is a performance issue and has not been prioritized. This could have been handled by the setModelState method. The uriParser object could have checked against the localStorage to see what properties actually had changed since the last saved state object. The state object returned from the makeAnchorMap offers for each dependent object a string representation of that object. This could have simplified the check against previous state objects if the states were stored as strings on the localStorage.

5.3.3 The additional responsibilities of the ToolApp

The toolApp has also been assigned responsibilities external to the initialization phase. This is because the toolApp already had been set up to listen to the "ready" event of the main iframe DOM. This DOM should not be confused with the DOM of the root window.

Mode discrepancy for GSuite tools. After setting the toolModel as described in a previous section the toolApp is responsible for deciding if the tool is a GSuite tool and if so hide the "isBasic" check box.

All GSuite tools are delivered with an "isBasic" check box for retrieving the other representation of the tool from the server. This is done by checking if the "isBasic" id is present. Only GSuite tools have an element with an "isBasic" id which makes this the identifying feature for GSuite tools.

To account for situations where the current mode of the HyperBrowser is not coherent with a newly loaded GSuite tool representation, the state of the sites mode needs to be adjusted accordingly. This is only possible to know as the tool is finished loading. The check to decide if the tool mode is coherent with the mode of the HyperBrowser is done by checking the isBasic checkbox against the modeModel object. This is why the toolApp needs to be injected with a modeModel upon initialization.

This situation can occur when rerunning a tool from the history section of the HyperBrowser. The items within this history section have functionality to rerun a tool. When a GSuite tool is rerun, its representation will still be in the mode it was when originally run. If the mode of the HyperBrowsers site has changed since running the tool, rerunning it will cause a mode discrepancy. The StateApp is set to change the mode of the whole site such that it is coherent with the mode of the rerun tool. It could be argued that the tool should be set according to the mode of the site instead. Be that as it may, that is a discussion outside the scope of this thesis.

Mode state discrepancies with the URI There are also scenarios where a tools mode and the HyperBrowsers mode are equal when the "ready" state is emitted but the URI is out of sync. This is the last condition the toolApp is responsible for checking before it returns.

The table 5.1 is a visual representation of the mode state of the Model, the GSuite tool, and the URI as the program traverses the code after a click on the mode button. The program algorithm for a mode change when a GSuite tool is loaded in the main iframe goes as follows:

1. The base case. The site is in basic mode. No code is being executed. The program is in listening modus.
2. A user clicks the basic/ advanced button.
3. The modeView catches the "click" event.
4. The modeModel is toggled to advanced mode by the modeView and triggers a "change:mode" event.
5. The modeCTRL listens to the "change:mode" event and checks the "isBasic" checkbox on the GSuite tools form element. It then triggers the "change" event on the iframe DOM.
6. The browser sets the location object of the iframe and requests the advanced representation of the tool from the server. A new item is added to the session history list.
7. The browser loads the new tool representation and emits a "ready" event.
8. The toolApp sets the toolModel when catching the "ready" event.
9. The toolModel is set and causes a session history replace. The previous "stateless" entry is overridden by this "statefull" entry.
10. The toolApp continues and sets the modeModel with the GSuite flag and triggers another mode change. This time, it sets the modeModel with the same mode it already has. The normal path of triggering the moving of the sidebar and checking the "isBasic" checkbox is this time derailed.
11. The mode change is now merged with the current location hash. Then session history is replaced with this new state object.

5.3.4 Models, views and controllers

The responsibilities of the models, the views, and the controllers are described here because they play a vital role when we discuss some important consequences of a design decision. The reasoning behind the choices made will be elaborated on in the discussion part of the thesis.

Step	Model	Tool	URI
1	Basic	Basic	Basic
4	Advanced	Basic	Basic
7	Advanced	Advanced	Basic
11	Advanced	Advanced	Advanced

Table 5.1: The state of different parts of the program while changing mode.

The Views

The StateApp has two different kinds of views, one for each state.

The modeView is responsible for defining the mode button and its drop-down menu located on the main navigation bar of the HyperBrowser site. It also listens for user click events on the mode buttons drop-down menu.

When initialized it is injected with the modeModel and can thereby toggle the mode directly on the modeModel object whenever a click event is caught.

To be updated when the modeModel is changed the view also listens to "change:mode" and "change:history" events. See section 5.4.2 for a discussion on the event naming convention. When these events are caught the view consults the modeModel directly to get its new state before re-rendering the mode button and its corresponding drop-down menu.

The toolView is somewhat different from the modeView in that it does not listen for input from the user. This is because all communication with the DOM within a toolView is done by the default HyperBrowser functionality. See section 6.3.3 for further reasoning.

The only time the toolView is re-rendered is when a new tool has been fetched from the server by the StateApp itself. On successfully fetching a tool from the server the toolModel emits a "change:tool" event that is caught by the toolView. This only occurs when the URI is changed due to a bookmark being loaded, a link within an external document has been clicked or a URI has been pasted into the browsers address bar or the back button of the browser is clicked.

The models

The StateApp is set up to handle two kinds of states, mode state, and tool state. These application states are stored on what has been labeled the modeModel and the toolModel. This is, however, a simplified picture. All model state is actually set by calling the model prototypes set method. The prototype responds with an event and it is the models responsibility to trigger the appropriate event according to whether the model was set from

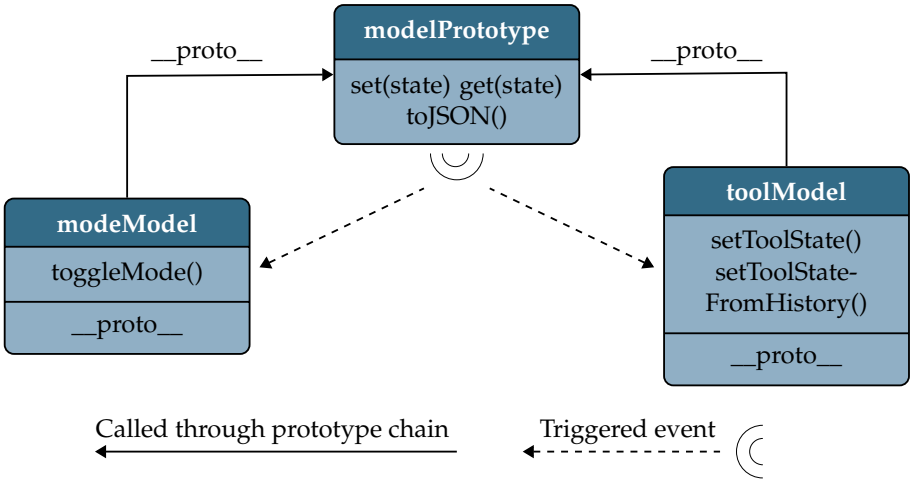


Figure 5.1: Communication between models and the model prototype.

the DOM or from a change in the URI. The problems of keeping a central unit for data storage like this will be discussed in section 6.3.1.

The toolModel has one additional responsibility: to decide if the state change was triggered from a GSuite tool or from a regular tool.

The Model prototype The model prototype is a common object shared by all models. This is where the shared functionality of the models reside. It contains code for setting and getting model state and for deleting state. There is also a method, the toJSON method, that returns all the state of the model in one object. See figure 5.1.

The controllers

The controllers are responsible for performing the specified state changes on the HyperBrowser site. The modeController toggles the sidebar when the mode changes and the toolController fetches new tools from the server. The latter is done by using the ajax method of the jQuery object. If fetching the tool is successful the toolController triggers a "change:tool" event.

The modeController is also responsible for controlling other mode-specific behavior when the mode changes. If the current tool is a GSuite tool, the modeController checks the "isBasic" checkbox and thereby requests a new GSuite tool representation from the server. For the DOM to notice that the checkbox has been checked by the program a DOM "change" event needs to be triggered. This is done by the modeController who utilizes jQuery to trigger the DOM event.

The modeController also sets the appropriate GSuite guide tabs when they are present in the toolFrames DOM. When the application changes modeState it is the modeControllers responsibility to make sure the correct GSuite tab is displayed. When in basic mode, the basic tab of the GSuite guides should be in front and conversely when the application

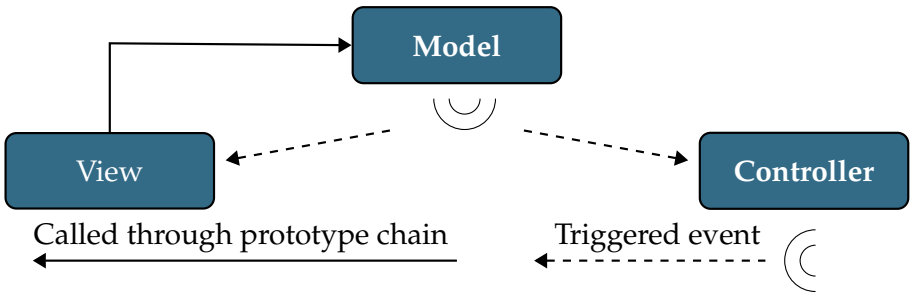


Figure 5.2: StateApp model, view, controller communication.

is in advanced mode the advanced tab of the GSuite guides should be displayed.

5.4 Communication between objects - Emulating the Backbone library

Some of the objects described in this section and their communication are inspired by the Backbone library objects (*Backbone.js* 2016). Specifically, the communication between the view and the model objects are emulated. See figure 5.2. The view handles input from the user and relays the input to the model. Since the view is injected with a specific model pointer it can do this by a regular method call on the model.

The model, on the other hand, is oblivious to the existence of the view. This enables many views to display the same data but in different ways. For example, displaying the same data in a Venn diagram and in a graph is made possible by utilizing events for communication between objects. Strictly speaking, this is not required the way the StateApp is built today. There wouldn't be a problem to tie the model to the view with direct method calls. The oblivious model is a feature for the future rather than a necessity of today. It follows the "open closed" principle mentioned in section 2.2.5 by letting the program be open to extension.

In addition to storing state, the StateApp model handles some logic. This is in accordance with how the backbone library implement their model. Giving the model responsibilities beyond that of storing state is contrary to a strict MVC thinking. The model is supposed to only have that one responsibility. The backbone way of constructing the model bears a close resemblance to the MVVM variety of the MVC pattern presented in the background chapter (2.3.4). Both the view and the model handle logic.

5.4.1 Naming of objects

In hindsight the naming of the model might not have been as concise as desirable. It would have been easier for readers of the code to get an immediate conception of the roles of the objects if the model were named according to the MVVM variety of the MVC architecture. That would better have conveyed the role as a mediator between data and presentation layers.

At the same time, the responsibilities of the model had gradually grown large and some of the functionality needed to be outsourced to an other object. Specifically all the action related methods such as opening and closing the side panel and communicating with the server could be done from a specialized object. This new object was called the controller because it was responsible for decisions concerning visual representations and information gathering actions.

This separation of concerns makes the program easier to extend and change, but is also somewhat more complex by adding extra items to keep track of.

5.4.2 Event naming conventions

To distinguish what object an event is emitted from, events are named with the type of the event, "set" or "change" event, and then the name of the triggering object. For instance, when a tool model triggers a change event it will be emitted as "change:tool". This enables views to listen only for the events triggered by "their" model.

Another solution which could have worked equally well could have been to let the views listen for general change events and compared the attached model to their injected model to see if the event concerned them. This could have resulted in a cleaner code base though with a small performance hit.

Although both of these naming conventions could have worked for the communication between views and models they would have made problems for communicating with the StateApps uriParser object. The initial idea was to let the StateApp uriParser object listen for general events. Whenever a model, be it a modeModel or a toolModel, triggered a "set" or "change" event the uriParser object could pick it up without having to hard code their names. The dispatcher had been created to emit general events whenever a specific event was triggered if there where listeners listening to general events so this scheme was already fully supported.

The problem with this approach was that there were situations where the uriParser object should not catch set and change events. For instance, every time a model was changed due to a change in the URI it would emit a "change" event to notify its listening views. This event would also have been picked up by the uriParser object resulting in resetting the URI and creating a looping condition.

To solve the problem, we decided to let the models emit a distinct event for notifying changes coming from changes within the application separating them from those coming from changes to the URI. This kept the models and uriParser object loosely coupled.

5.5 The dispatcher prototype

The StateApp dispatcher object is the central object for object communication. The main purpose is to provide a system for communicating state

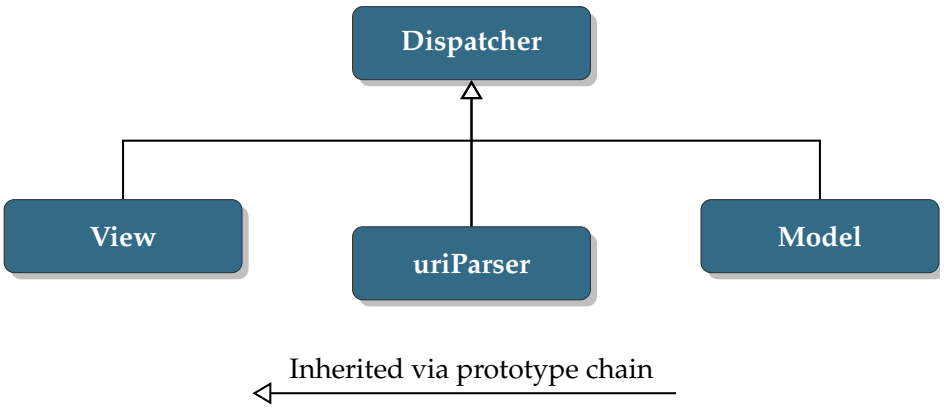


Figure 5.3: All the defined prototype objects have the dispatcher object as their prototype. This means that all these objects have available all the functionality of the dispatcher prototype object.

change between the participating objects. It is modeled after the Observer pattern but the implementation is somewhat different than the classical versions described in the foundations chapter(2.2.5). Instead of maintaining a list of pointers to Observers, the dispatcher object maintains a list of functions to be called. This deviates from the "classical" implementation of the Observer pattern:

Since JavaScripts offers functions to be passed around as objects, as explained in 2.2.3, it can register a "callback" function to be "called" when the specified event is triggered. With the "classical" Observer pattern the listener only registers an object pointer pointing back to itself. The Subject leaves to the Observer to decide if it wants to call for the updated state.

With the StateApp the triggering object (the Subject) passes along a pointer to itself for the receiving objects (the Observers) to use if necessary. When the dispatcher is prompted to trigger or dispatch an event it goes through the lists of callback functions and calls them directly. An attached context pointer is registered together with the function. This context pointer represents the object to which the functions "this" keyword points. This way the program execution can address properties of the context object as if it was calling the property through the object itself.

Since JavaScript is a loosely typed language it does not offer safety constructs ensuring that the implementing classes maintain the contract of the interfaces. This causes a strict implementation of classical design patterns like the Observer pattern unprofitable. Instead, the dispatcher can just call some function with the attached context pointer and go on being Subject.

To bind the callback functions to the context pointers, the dispatcher object utilizes the call function of the Function prototype described in section 2.2.3.

Chapter 6

Discussion: On applying state management

As the Internet has matured, we, the users of the World Wide Web, have started to expect certain behavior from a website (Michael Mikowski and Powell 2014, p.85). Whether the site is about displaying dynamically rendered news articles, or as is the case of the HyperBrowser, doing statistical analysis on a genome, the users expect to be able to use the built-in actions most browsers provide. Going back and forth using the navigation arrows or bookmarking a page for later reference or even sending a link to a friend or colleague is built-in functionality every web site should strive to provide for its users. This latter point should naturally be avoided by private applications where users are required to login. These application should not encode the local state onto the URI for external sharing of links. Nevertheless, they should also strive to enable the other expected browser behaviors.

Providing this functionality is, however, not as easy as it may sound when it comes to extending an existing, "thin-client" application. This chapter will discuss the lessons learned while implementing these user anticipated browser behaviors on the specific use case of the HyperBrowser.

6.1 The HTML5 History API is not sufficient to enable state sharing

There exists a number of JavaScript frameworks and libraries promising history management. So also the browser-native HTML5 History API with its pushState method and the popstate evnet. Nevertheless, these frameworks can only cater for local state management. For instance, the HTML5 History API offers a way to store state in objects. This enables storing state local to the browser. When saving state with the pushState or replaceState methods the History API stores the state object within the local resources of the current browser as explained in the background chapter (2.2.1).

There is only one problem: It does not enable sharing URI's. The

history object is by the W3C called the "session history" implying that the history will not survive the session (*Session history and navigation* 2010). As mentioned earlier the state of a page needs to be encoded in the URI to be shared. When using the History API the state is encoded in the URI by means of the conventions of that particular application but the actual state is never stored in the URI. This renders the History Api unfitted for external state management.

Although using the History API is perfect for enabling the default browser behaviors of the navigation buttons and the bookmark, other means must be used for sharing the current state with other browsers.

6.1.1 The HyperBrowser server is not enabled for using the HTML5 History API

In the case of the HyperBrowser, the History API could not be used for enabling the default browser behavior. Every time the URI changes, whether it is due to a back button click, opening a bookmark or pasting in the URI in the browsers address bar, the browser requests the server for a resource. If the resource is stored using the HTML5 History API the resource is located on the browser and the server does not know that particular URI. That is why the server needs to be enabled for using the History API. It needs to accept any URI request coming from a browser, also those not registered on the server and redirect them back to an existing resource, for instance, the index.html page. If this is not done the server will respond with an error message.

According to the HyperBrowser technical staff this was not possible due to the double layer of servers where the receiving Apache server was only functioning as a relay to the underlying Galaxy server which they did not know how to control. Be that as it may the local state needed to be encoded onto the URI both to enable browser functionality and to enable sharing state.

6.2 The uriAnchor as URI encoder. A good choice?

The choice of using the uriAnchor library to encode application state onto the URI might not have been a good choice. To depend on a library which add functionality already provided by other parts of the program might seem redundant.

It leads to more complex code by adding extra variables. It will also add to the download cost as it adds to the amount of code needed to run the program on the client. For instance, using the jQuery param method in combination with the default JavaScript encodeUriCompontent will cater for the stringification process. The only problem is that jQuery does not offer a method to parse a URI to an object.

JSON is an alternative with JSON.parse(). JSON offers both a parse and a stringify method. The stringify method however does not allow to stringify objects with internal objects.

The uriAnhor library provides the possibility to handle complex objects (5.3.1). Complex in this context means that the named properties of the object to be stringified can have dependent variables associated with them. These dependent variables will be encoded on the fragment part of the URI along with all the independent variables.

This is a good fit for the StateApp. It provides the ability to parse and stringify a state object. The mode state only needs a string representation whereas the tool state need more complex data structures representing it.

A natural question to ask here is why couldn't the value of the independent variable just be a regular object and the string value associated with the independent variable be stored alongside the other dependent variables inside the value object like listing 6.1?

```
$.uriAnchor.setAnchor({
  mode   : 'advanced',
  tool   : {
   toolName : 'Analyze genomic tracks',
   serializedForm    : 'dbkey=blackrust',
   currentSelection : 'galaxy_main'
  },
  otherFutureProperties : 'red'
});
```

Listing 6.1: Example of a stringified complex object.

The answer is that by introducing an explicit dependency relationship, the association between dependent and independent variables could be visible in the URI. As shown in listing 5.5 the relationship is indicated by a colon (:). Without the visible dependency relation it would be hard to express the relationship in the URI.

There exists other less official code on the World Wide Web that provides this functionality but that code is usually not maintained and might cause problems in the future.

6.2.1 The complications of using the uriAnchor

The decision to use the uriAnchor as a means to stringify and encode state objects onto the address field of the browser turned out to be more troublesome than anticipated. In early stages of developing the StateApp there existed a need to retrieve a tool from the server initiated by the StateApp. This had to be done by sending a predefined property, GALAXY_URL, attached to the search part of the URI. The value of this property should contain the full URI:

```
https://HyperBrowser.uio.no/state/hyper?GALAXY_URL=https%3A//
    HyperBrowser.uio.no/state/tool_runner&tool_id=hb_test_1
```

Listing 6.2: The full URI to retrieve a HyperBrowser tool.

It turns out that the uriAnchor library decodes the URI before parsing it. This is problematic in relation to URI syntactics (2.2.2). The URI grammar restricts the use of the colon (:) as it is used to separate the protocol scheme from the rest of the URI. Two examples are http: and ftp:. Because

the uriAnchor decodes the URI before parsing it and by that expose any percent-encoded characters, the colon indicating the relationship between dependent and independent variable breaks any web addresses stringified in the URI.

Percent-encoding enables the use of restricted characters in a URI. This is also called URL encoding. The percent sign is used as an escape character followed by a pair of hexadecimal digits representing the characters ASCII byte value (*Percent-encoding* 2016). The colon is thus encoded %3A as seen in listing 6.2.

When the uriAnchor parser meets the search part of the URI where the tool address is stored in the GALAXY_URL parameter it takes the colon in https:// to be an indicator of the dependency relationship and thus breaks the address.

To remedy this problem the default dependent/ independent relationship indicator, the colon, can be changed. Strangely enough, the uriAnchor library provides the possibility to change the default delimiters but the code that does this is disabled. The documentation outlines the default delimiters and explains how to replace them but does not mention that this possibility is disabled and thereby does not work. To enable replacing delimiters one needs to read and understand the whole code base of the library. It became a challenging hunt for fixing the bug. When found, a dash followed by a greater than sign "->"where chosen to indicate the dependency relationship.

In hindsight a better solution might have been to develop the parse and stringify code needed by own means. Though developing the functionality form scratch could have saved the time spent debugging third-party code it would also have consumed from the scarce time resources available. This is always a gamble when using third-party code. The time spent researching the options, learning to use, and in the rare cases debugging the software is time consuming and not always worth it. This is however not easy to estimate upfront.

6.3 Driving application state: The URI or the state model?

The problem of where to contain the business data on the client was introduced in section 2.3.3 of the background chapter. The Backbone library with all the other MV* JavaScript frameworks and libraries provide a model for keeping the application state. This model will function as the central location to drive synchronization. Any part of the program in need of synchronizing consults the model to do so. Backbone declares that: "The single most important thing that Backbone can help you with is keeping your business logic separate from your user interface" (*Backbone.js* 2016). When building the StateApp for the HyperBrowser this was the main benefit when emulating the Backbone library, to discern presentation in the view and keep the data in the model.

6.3.1 The complications of keeping a central model

The MVC architecture suggests to keep all business data in central models. This way there will always be only one source of truth. All changes must go through the model to be realized.

In the case of the StateApp a disadvantage of depending on a central model is that both state changes coming from user input and from URI "hashchange" events need to be registered on the central model. At the same time, the user activity also needs to be added to the URI whereas changes to the URI should not float back down to the model. An example of this problem can be sketched as follows:

1. A user toggles the mode button from basic to advanced state.
2. The central model is set with the new advanced state.
3. The stateParser sets the URI with the new state. This changes the fragment identifier.
4. The browser detects the change on the fragment identifier and emits the "hashchange" event because the fragment identifier was changed.
5. The central model is set again.

If not detected this behavior will cause a looping condition. See figure 6.1. Fixing this problem with conditionals is not very hard but leads to an unnecessarily cluttered code base. This may lead to slower development and is more prone to bugs as stated by Mikowski et al. (Michael Mikowski and Powell 2014, p. 86).

6.3.2 The Anchor Interface pattern as a substitute to the central model

The anchor interface pattern suggested by Mikowski et al. proposes to use the URI as the driving force behind every state change. Setting the URI before anything else enable the URI to function as a central resource to be consulted when parts of the application need to synchronize. See figure 6.2. This solves the problem of the loop condition just described. This way of setting the URI directly takes advantage of the automatic "hashchange" event emitted by the browser whenever the fragment identifier is changed instead of being forced to circumvent the effect of this feature.

Notwithstanding these benefits, adding state onto the URI directly by the receiving event handlers also poses a problem with the browser session history management. Sometimes the session history entry needs to be delayed to maintain a correct representation and avoid the problem of double session history entries. This will be shown and the problem will be discussed in the following sections.

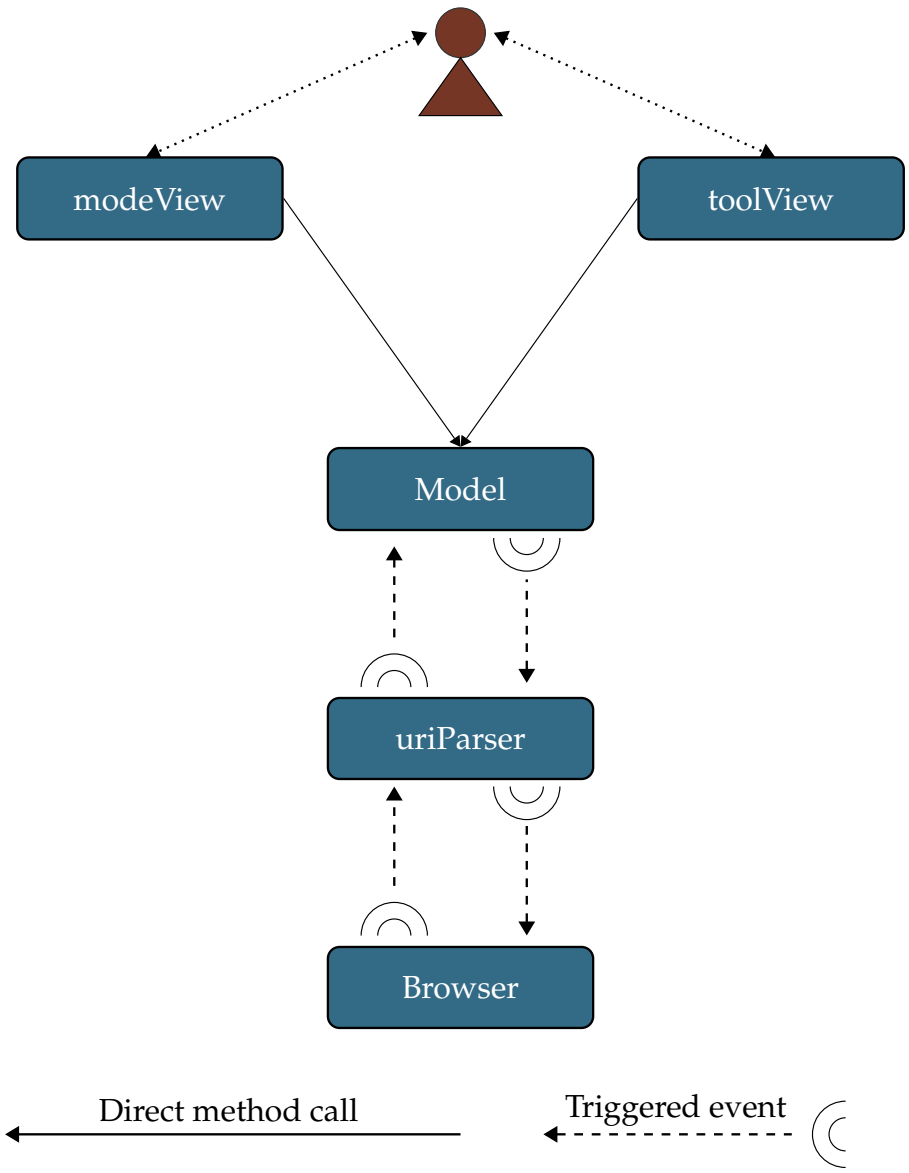


Figure 6.1: The central model. Looping condition.

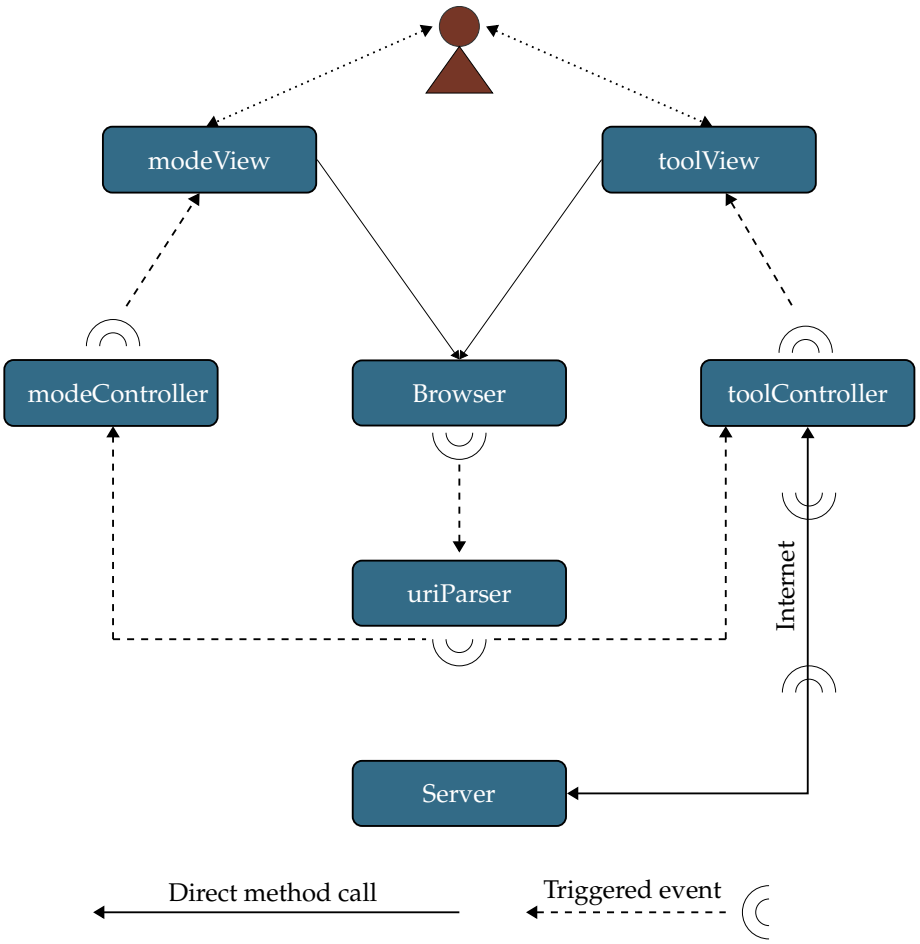


Figure 6.2: The Anchor Interface pattern if implemented on the stateApp.

6.3.3 Hijacking functionality or passively recording state?

When applying state management onto an existing page, one needs to consider the pros and cons of hijacking existing functionality, versus merely recording the states passively as they emerge. This consideration needs to be investigated because of the way the session history management works. Interfering with existing server communication can be a solution to the problem of double history entries when managing state. We start off by presenting the default browser behavior when a user clicks a regular link in a document and then introduce the problem:

When a user clicks a link on a web page, the default browser behavior adds the URI of the link to the browser's address field, and records an entry to the session history before requesting that page from the server. As the response returns the browsers caching engine stores the received data and sends the response to the browsers rendering engine who displays the page on screen. This is straight forward for static pages.

The problem of the double history entry I: Introducing client-side states, like a basic and advanced mode, can make things more complicated. As proposed in a previous section such states can be stored on the fragment identifier and as they are, an additional history item is added to the session history. This means that for every new type of state to be recorded there will be an additional history item added. To navigate to the previous site one now need to click the back button twice. This might not seem like a big problem but is nevertheless an unexpected behavior that might confuse the user.

An example of this could have happened when opening the Hyper-Browser for the first time. The browser loads the page and adds a session history item. Then the StateApp consults the localStorage and sets the state stored there or if no state is stored initializes the application with the "basic" mode. This creates a second browser history entry without the user knowing. If not dealt with this would have forced the user to click the back button twice if she wanted to go back to the previous page.

Hijacking functionality

To be in full control of when the browser adds an item to the session history it will be necessary to interfere with the existing server communication. This includes preventing the default behavior of the web browser. Again, for a static page this wouldn't be much of a problem.

An example of overriding the default behavior of the browser is to control the behavior of a link. After preventing the default behavior with the preventDefault() method of the event object, emitted by the browser with every event, one could get hold of the value of the links source attribute and then use the jQuery ajax method to get the document from the server. After appending the received document to the DOM, the state of the new page could be stringified along with any local state and then first appended to the fragment identifier. After building the URI this way

one could set the location with this new URI and thereby setting the history only once.

Hijacking the default behavior like this might not be as straight forward with an existing application. For instance, when doing a selection in one of the HyperBrowser tools there is a lot of complex assessments, server communication and validation going on. To interfere with this kind of code the front end-developer should be familiar with the inner workings and expectations of the server. To tamper with it could lead to unknown behavior and unimaginable consequences. When dealing with complex functionality like this it might be better to let the existing code do its thing and focus on recording the state passively as it emerges.

Another important objection of hijacking the existing functionality in the case of the HyperBrowser is that the state of the tool can not be anticipated until the tool is received from the server and rendered readable onto the DOM. This means that the stringified tool state must be added to the URI after the browser has loaded the document and added a history entry. Interfering with the click event before the tool state is being requested from the server would only enable recording the tool state already present, not the tool state about to be requested.

At the time of implementing the StateApp these arguments were found sound. Both hijacking the existing functionality delivered from the HyperBrowser and hijacking the default submit behavior of the browsers form element were disqualified for the over stated reasons.

This decision turned out to be a disadvantage because it was based on the faulty assertion of the latter argument. When using the default browser behavior of submitting a form, the server will respond with a new document. This is what causes the browser to add a history item. If the form is hijacked when submitted, the serialization of the form can be sent as an Ajax request. Because of the servers "content negotiation" scheme, the Ajax request is responded to with the same format as was sent with the request. For a serialized form this is a string. The string can be parsed and written onto a document created by the local program and added to the DOM without the browsers history knowing. The form can then be serialized again and added to the history causing only one history entry.

Although the last argument turns out to be based on the wrong assumptions the first argument still holds. It is a difficult task to analyze the existing code to be sure it is not corrupted if overridden. Therefor it is also necessary to analyze the other option of passively recording state.

To passively record state

The problem of the double history entry. Revisited. Instead of overriding the existing handling of server communication one can record the different states as they are set. For instance, every time a user does a selection or otherwise alter a tool form on the HyperBrowser site, the server is consulted and a new state of the tool form is sent from the server. When the browser is done rendering the DOM it fires the documents "ready"

event. The StateApp can then serialize the form and save the resulting string to the URI. This means that for every state change the server first adds a session history, then requests the new tool state from the server, renders the new tool state and then fires the documents "ready" event. The StateApp adds this new state to the URI and the browser adds another history item, resulting in the double history entry problem once again.

6.3.4 Conceal the double history entry by replacing the URI

The obvious solution to the problem of the double history entry is to use the replace method of the location object. This is a method present on the location object by default in most browsers. When setting the location, instead of adding a new history item on top of the existing history entries it deletes the previous entry before adding the new. This works fine for the scenario described in the previous section. The server adds a history item when it requests the document and the StateApp replace that entry with the serialized version of the new tool form. Only one entry has been recorded.

The problem of triple history entry: This, however, does not render an adequate solution when opening a new tool from the tool menu or the GSuite guides of the HyperBrowser. It literally adds to the problem by adding an extra event. Because the name of a tool is not present in the serialization of a tool form, the tool name must be recorded when clicking the tool link.

The scenario goes as follows:

1. The user clicks a tool, either in the tool menu section or from the GSuite guides. This is recorded by the toolModel and then added to the URI and a history entry is recorded by the browser.

2. At the same time a request is sent to the server and another history item is added to the session history.

3. The server responds with a tool in its initial state and as the browser finishes rendering the DOM the StateApp records the tool state and adds it to the URI causing yet another entry being added to the session history.

This results in three history items being added when opening a tool.

A simple replace will not work here. Even though the last two history items follow the same pattern as described above, the first recording, that of the tool name, does not. This entry will replace the history item already present and thereby removing the previous state. That ruins the intentions of the back button.

6.3.5 Delay adding state to the URI

A solution to the problem of triple history entries: The solution to this triple history entry, suggested by the StateApp, is to add the name of the tool to the toolModel but stop the name from being added to the URI. When the tool state is set on the toolModel after the browser has rendered the new tool representation received from the server, the tool name is already recorded on the toolModel and is thereby added to the URI with the rest of the tool state. Following the pattern described above, this last history entry will replace the entry made by the browser when the server was requested for the state change.

Benefit of the central model compared to using the URI as a state driver

This is an example of the benefits of having a central model on which to store state instead of depending on the fragment identifier as the only state storage location. When depending on the fragment identifier for state storage there will be added a history item for every time state needs to be saved. This is not something the programmer can control. Using a central model on which to store state, that does not influence the session history, gives the developer more freedom.

Mode state changes can not replace the previous session history entry

The problem of the single history entry: The pattern described above only works because the browser adds an extra history entry when the server is requested for state change. This entry is an empty entry when it comes to storing state. It is not visible in the browsers address field. This is because the address field belongs to the main window URI. The empty entry is set on the location object of an iframe and is thus not visible to the user. More on iframe issues in section 2.2.6. Replacing this entry restores the expected back button behavior when a tool updates its state, but does not do so when the mode is changed on the HyperBrowser.

Mode only sets one history entry. Replacing the previous entry on the session history deletes the previous history and thereby prevents the back button to lead back to the previous state.

Solutions to the single history entry problem: A solution to the problem is to add a conditional statement deciding whether the entry comes from a mode change or a tool state change. If mode should be set use the assign method of the location object, otherwise use the replace method. Although this solves the problem of deleting the previous state it introduces another problem: that of deciding if the state change was caused by a mode change or a tool state change.

One way to discern whether the change is a mode change or a tool state change is to name the models. This is the solution used by the StateApp but introduces the problem of tight coupling between objects. Remember how all state change go through a model object before the specific state event is

emitted. Naming the models for identification in other objects forces the receiving object to hard code the model names.

The tight coupling can be handled by defining callback methods to pass along with the models "change" event such that the StateApps stateParser object, for example, just calls the callback defined in the current model. There is, however, a problem with this solution also. An extra condition is needed when changing mode when the application displays a GSuite tool.

The dual mode of the GSuite tools introduces an extra history entry

Because the tools contained in the GSuite project has different representations for the two modes, one basic and one advanced representation, they can not follow the same conditions as other mode changes. When a GSuite tool is open and the mode of the website is toggled the server needs to be requested for the opposite representation of the tool. This results in the same flow as for a regular tool state change except that the mode is also set. This means that for every mode change on a GSuite tool tree history entries are set. First, the mode causes a history entry, then the server causes another and then when the new tool state is set on the fragment identifier this causes a third history entry.

A solution to changing mode from within a GSuite tool: For the StateApp we have chosen a solution where setting the first mode state is delayed till after the tool state has replaced the history entry caused by the request for a new tool representation. This is the same method as used when the setting of a tool name is delayed and merged with the new tool state after it has returned from the server. In addition, the mode model is marked, adding a property with the name GSuite, and then added to the replace condition when setting the URI. For implementation details see the results chapter (5.3.3).

Replacing session history breaks the browsers default caching system. When replacing browser added session history entries with local state changes onto the fragment identifier, the default caching system of the browser is broken. Since the browser-added history entry for receiving the document from the server is overridden by a change on the fragment identifier, caching will not be consulted.

The remedy is already present in the program: the program already is programmed to fetch the tool state from the server when a bookmark or otherwise a legal URI is pasted onto the browsers address bar. Unfortunately, this only works when going back to another tool state. Going back to the GSuite guides will not work because there was never a session history recording of the welcome page being received from the server. This history entry was overridden with setting the mode state upon initialization or pushed back to make place for the empty hash entry seen through the eyes of the browser cache. The browsers caching functionality does not work and the Ajax method is not set up to request the GSuite

welcome page. The extended use of iframes as content separators pose a problem and will be discussed shortly, in section 6.3.9. NB!!!

6.3.6 Disqualifying the Anchor interface pattern

The need to delay adding the mode change as seen in section 6.3.5 disqualifies the Anchor interface pattern as being an alternative to data-binding using a central model. The central model is needed to temporarily store the mode state while waiting for the tool state to return from the server. Then the two states can be added to the URI simultaneously. The Anchor interface pattern states that user events should be recorded on the URI without any other logic being performed.

6.3.7 Inconsistencies between browsers session history implementation

The previous discussion also shows that passively recording state leads to many conditionals and possibly coupling the objects too tight. The worst part is that passive recording scheme doesn't work across all browsers. It turns out that only the Google Chrome, Opera, and the Mozilla Firefox browsers register hash changes as session history entries.

Microsoft's Internet Explorer browser and Apples Safari browser does not register additions to the fragment identifier as session history which forces the browser back to the GSuite guides every time the back button is clicked. This might not be all bad. The Chrome, Opera and Firefox browsers need a hack to get back to the GSuite guides as described in the next paragraph.

6.3.8 Deciding if the "hashchange" was due to a back button click

The StateApp history object is programmed to detect whether a "hashchange" event is caused by a back button click or other URI changes. Detecting a back button click is not a feature offered by the browser. To our knowledge no browser implementation differentiate whether the "hashchange" event was caused by retrieving a bookmark, pasting a new URI to the browsers address field, coming from an external link or from a back button click. The URI's are all just URI's and thereby equally built. Without this being differentiated by the browser the developer is left in the dark.

Nevertheless, deciding what caused the URI change can be done by keeping the current state and the previous state on localStorage and then compare the new URI with the previous entry on localStorage. If they are equal we have a back button click.

Hack to get back to the GSuite guides As seen earlier, using the location.replace() method to override the server added session history entry, the recording of receiving the GSuite guides has been overridden by local mode changes. This means that there are no record of the GSuite

guides in the browsers cache and they must be requested from the server. When moving between tool states it is not relevant what caused the back behavior. They are all just tool states to be fetched from the server using Ajax. When going back to the GSuite guides however the server must be requested for the welcome.html document specifically. This is the document that contains the GSuite guides.

The way the StateApp detects if it is going back to the GSuite guides is by checking the localStorage and see if the previous tool state is set. If it is set on local storage but not new URI, the application should request the server for the welcome.html document.

Fetching tools with Ajax from outside an iframe has complications as will be shown in the next section.

6.3.9 Ajax calls from outside the document of the iframe create problems

Every time a selection is done within a HyperBrowser tool a new document is requested from the server. The received document contains all HTML markup for the tool along with several JavaScript functions. The markup consists mostly of a form element and all the select options contained within that tool. The JavaScript functions are global functions that are added to the window object of the iframe the document is targeted for. These functions do a lot of server communication, for instance, validating if the selections form valid hypothesis or otherwise are possible to evaluate. The communication with the server is done using the jQuery ajax method. The ajax method needs a location to address the request and provides the URL property for this. When the URL property of the ajax method has been provided with only a relative URL, as is the case for the tool forms of the HyperBrowser, the URL property uses the base URL relative to the document on which the code is present. Every document returned from the server this way has gotten the name "hyper" which renders the base url "http://HyperBrowser.uio.no/path/hyper".

This works well when the code is called from within the document. When the code is received by other means,, for instance, after parsing the URI of a bookmark, the relative URL does not contain the name of the document. When parsing a URI coming from a bookmark, linked to from an external document or is pasted into the browser's address field the program parses the URI and at some point an ajax call is produced. In contrast to when HTML forms are submitted the ajax way of getting data from a server is not returned in the form of a named document. The information is returned in some other format, it could be XML, JSON or in this case, a regular string. This string is then written onto the correct iframes document. This iframe document has not gotten a name. The relative URL of this document will be 'http://HyperBrowser.uio.no/"path"/"'. Since the document has no name the name section of the URI will be omitted. This breaks all the tool functionality.

The solution has been to add the "hyper" name to all the ajax calls in the

received string. This is a hack and will break if the name of the document is changed.

The usage of iframes stems from the fact that the HyperBrowser is an extension of another open source web-based platform, for data intensive biomedical research, the Galaxy platform. The HyperBrowser is dependent on how the Galaxy project was originally implemented. The consequences of having to deal with iframes for the StateApp will be discussed later in this chapter.

6.3.10 How should a back button behave anyway?

There are probably many opinions on how the back button should behave in the context of a browser. Should every little change in a form be recorded and made "backable" or should a back click only lead back to a different URI. One suggestion is to always go back to the last perceived state (Holst 2014). Because of the relative easiness of changing a selection or other form element there is little point to use the back button to undo a selection. Then the cursor must be moved all the way up to the back button to go back. It is as easy or even easier to do the selection again.

When it comes to how the back button should behave on the HyperBrowser web site, the question is whether to go back displaying the previous tool state or going back to the GSuite guides? We have chosen to let every state change be recorded and thereby be "backable".

To establish what comprises the last perceived state, real users must be consulted. Microsoft and Apple has chosen to take a stand and not allow changes to the fragment identifier to be "backable". This choice might have been done on the grounds of user testing or been based on the research presented in the background chapter. Either way, disqualifying the application developer this way constrains initiatives like the current one, where encoding state on the URI has been done to reintroduce the URI as a unique identifier for all client-side state.

Chapter 7

Conclusion

7.1 Summary

This thesis has been discussing client-side state management in the context of two legacy applications—the browser and a server-generated, thin-client website. By implementing a working state management solution, the challenges of enabling the default browser functionalities of the back and forward buttons, the bookmarking feature, and shareable state has been examined.

The assertion that state must be encoded onto the URI for it to be sharable has been promoted along with discussions concerning how to do so. Concrete organizational schemes has been considered. Using the URI as state driver has been evaluated against keeping a central model for synchronization between architectural layers of the application.

General considerations of the browser's default implementation has been brought up side by side with particular challenges of the Hyper-Browser. To passively record state, without hijacking existing means of retrieving state changes, has been shown to be laborious. At the same time, taking over existing functionality can be daunting, especially for junior developers without necessary confidence, both in client-side programming, and in the specifics of the server application.

The browsers internal history management API has been tested and found not to satisfy the demands for sharing client-side state. Using the pushState method together with the the "popstate" event only stores state local to the current browser. This enables implementation of the default browser functionality but not of sharing state. Some legacy applications do not have the means to facilitate these methods for state management and is thereby left to utilize the fragment identifier for such ends.

Microsoft and Apple has decided not to register changes to the fragment as history events. Though still able to facilitate the sharing of state, the discussion concludes by indicating that this decision precludes internal state management for certain legacy application and thereby disables the back and forward buttons for these applications.

7.2 Thoughts on future development

The decline in use of the default browser functionalities, as presented in the background chapter, may stem from the lack of inconsistent implementations of these functionalities. It might also be a trend that is caused by the general transformation of the World-wide web into a network of complex and highly dynamic applications where the default browser behaviors are outdated. Answering these questions might give clarity and promote tools for developing sophisticated and ultra-modern, social web software, it might also point in the direction of the need for browser features that makes state management easier for the developers.

Even though many modern web applications are aggregations of content that are either personal or of lesser interest to the public, there are still many websites that would benefit from handling state. Both for maintaining the default browser functionality but also for the sake of sharing knowledge and thereby continue the augmentation of the human intellect as was the dream of the Internet pioneers.

Bibliography

Aimar, Alberto et al. (1995). "WebLinker, a tool for managing {WWW} cross-references". *Computer Networks and {ISDN} Systems* 28.1–2. Selected Papers from the Second World-Wide Web Conference, pp. 99–107 (cit. on p. 23).

Anand, Vishal. and Deepanker. Saxena (2013). "Comparative study of modern web browsers based on their performance and evolution". *Computational Intelligence and Computing Research (ICCIC), 2013 IEEE International Conference on*, pp. 1–5 (cit. on p. 12).

Angularjs (2016). Webpage.
https://angularjs.org/
(visited on 04/25/2016) (cit. on p. 24).

Architecture of the World Wide Web, Volume One (2016). Webpage.
https://www.w3.org/TR/webarch/#identification
(visited on 04/10/2016) (cit. on p. 19).

Arno, David (2010). *Why JavaScript is a toy language*.
http://www.davidarno.org/2010/05/18/why-javascript-is-a-toy-language/
(visited on 02/13/2016) (cit. on p. 12).

Backbone.js (2016). Webpage.
http://backbonejs.org/
(visited on 04/18/2016) (cit. on pp. 24, 49, 56).

Barth, Adam, Collin Jackson, and John C. Mitchell (2009). "Securing Frame Communication in Browsers". *Communications of the ACM* 52.6, pp. 83–91 (cit. on p. 23).

Berners-Lee, Tim, Robert Cailliau, Jean-François Groff, and Bernd Pollermann (2010). "World-wide web: the information universe". *Internet Research* 20.4, pp. 461–471 (cit. on p. 27).

Berners-Lee, Tim, Roy T. Fielding, and Larry Masinter (1998). *Uniform Resource Identifiers (URI): Generic Syntax*. RFC 2396. RFC Editor.
http://www.ietf.org/rfc/rfc2396.txt
(cit. on pp. 1, 10, 11).

Castro, Paul, Frederique Giraud, Ravi Konuru, John Ponzo, and Jerome White (2006). "Before-Commit Client State Management Services for AJAX Applications". *2006 1st IEEE Workshop on Hot Topics in Web Systems and Technologies*. IEEE. Boston, MA, USA, pp. 1–12 (cit. on pp. 2, 24).

Catledge, Lara D. and James E. Pitkow (1995). "Characterizing browsing strategies in the World-Wide web". *Computer Networks and {ISDN}*

Systems 27.6. Proceedings of the Third International World-Wide Web Conference, pp. 1065–1073 (cit. on p. 2).

Chao, Johaph T., Kevin R. Parker, and Bill Davey (2013). "Navigating the Framework Jungle for Teaching Web Application Development". *Issues in Informing Science and Information Technology* 10, pp. 95–109 (cit. on p. 24).

Cockburn, Andy and Steve Jones (1996). "Which way now? Analysing and easing inadequacies in {WWW} navigation". *International Journal of Human-Computer Studies* 45.1, pp. 105–129 (cit. on p. 22).

Cockburn, Andy, Bruce McKenzie, and Michal Jasonsmith (2002). "Pushing back: evaluating a new behaviour for the back and forward buttons in web browsers". *International Journal of Human-Computer Studies* 57.5, pp. 397–414 (cit. on pp. 2, 22).

Comparison of layout engines HTML5 (2016). Webpage. https://en.wikipedia.org/wiki/Comparison_of_layout_engines_HTML5 (visited on 03/17/2016) (cit. on p. 17).

Crockford, Douglas (2008). *JavaScript:The good parts*. First Edition. Sebastopol, CA 95472: O'Reilly Media, Inc. (cit. on p. 3).

Document Object Model (DOM) Level 2 Events Specification (2000). Webpage. https://www.w3.org/TR/DOM-Level-2-Events/ (visited on 02/09/2016) (cit. on p. 15).

Flanagan, David (2011). *JavaScript: The Definitive Guide*. 6th ed. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media (cit. on pp. 8, 12, 15, 18, 42).

Foster, Ian, Savas Parastatidis, Paul Watson, and Mark Mckeown (2008). "How Do I Model State?: Let Me Count the Ways". *Communications of the ACM* 51.9, pp. 34–41 (cit. on pp. 7, 22).

Gamma, Erich, Richard Helm, Ralph Johnson, and Jules Vlissides (1996). *Design Patterns : Elements of Reusable Object-Oriented Software*. Eight Printing. Addison Wesley Publishing Company, Inc. (cit. on pp. 15, 16).

Grosskurth, Alan and Michael W Godfrey (2005). "A reference architecture for web browsers". *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. IEEE, pp. 661–664 (cit. on p. 8).

Helland, Pat (2005). "Data on the Outside Versus Data on the Inside." *CIDR*, pp. 144–153 (cit. on p. 22).

Holst, Christian (2014). *4 Design Patterns That Violate Back-Button Expectations*. Webpage. http://baymard.com/blog/back-button-expectations (visited on 04/30/2016) (cit. on p. 67).

Jackson, Collin and Helen J. Wang (2007). "Subspace: Secure Cross-domain Communication for Web Mashups". *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. Banff, Alberta, Canada: ACM, pp. 611–620 (cit. on p. 2).

jQuery (2016). Webpage. https://en.wikipedia.org/wiki/JQuery (visited on 02/09/2016) (cit. on p. 16).

jQuery reigns as top JavaScript library (2014). Webpage. http://www.infoworld.com/article/2861081/javascript/jquery-reigns-as-

top-javascript-library.html
(visited on 02/09/2016) (cit. on p. 16).
jQuery Usage Statistics (2016). Webpage.
http://trends.builtwith.com/javascript/jQuery
(visited on 02/09/2016) (cit. on p. 16).
Kannan, Natarajan and Toufeeq Hussain (2006). "Live URLs: Breathing Life into URLs". *Proceedings of the 15th International Conference on World Wide Web*. WWW '06. Edinburgh, Scotland: ACM, pp. 879–880 (cit. on p. 23).
Krug, Steve (2005). *Don't make me think: A common sense approach to web usability*. Pearson Education India (cit. on p. 31).
Mikowski, Michael and Josh Powell (2014). *Single Page Web Applications*. 20 Baldwin Road, Shelter Island, NY 11964: Manning Publications Co. (cit. on pp. 23, 31, 53, 57).
Mikowski, Mike (2016). *UriAnchor*. Webpage.
https://github.com/mmikowski/urianchor
(visited on 03/28/2016) (cit. on p. 23).
Nielsen, Jacob (1996). *Why Frames Suck (Most of the Time)*. Webpage.
https://www.nngroup.com/articles/why-frames-suck-most-of-the-time/
(visited on 04/10/2016) (cit. on pp. 1, 20).
Nielsen, Jakob (1997). "User interface design for the WWW". *CHI'97 Extended Abstracts on Human Factors in Computing Systems*. ACM, pp. 140–141 (cit. on pp. 19, 31).
nodejs (2016). Webpage.
https://nodejs.org/en/
(visited on 04/21/2016) (cit. on p. 34).
npm (2016). Webpage.
https://www.npmjs.com/
(visited on 04/21/2016) (cit. on p. 34).
Obendorf, Hartmut, Harald Weinreich, Eelco Herder, and Matthias Mayer (2007). "Web Page Revisitation Revisited: Implications of a Long-term Click-stream Study of Browser Usage". *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '07. San Jose, California, USA: ACM, pp. 597–606 (cit. on pp. 2, 22).
Orchard, David (2006). *State in Web application design*. Tech. rep. The W3C technical architecture group.
https://www.w3.org/2001/tag/doc/state.html
(cit. on p. 7).
Percent-encoding (2016). Webpage.
https://en.wikipedia.org/wiki/Comet_(programming)
(visited on 03/29/2016) (cit. on p. 56).
Rheingold, Howard (1985). *Tools for thought*. Webpage.
http://www.rheingold.com/texts/tft/
(visited on 04/21/2016) (cit. on p. 27).
Sandve, Geir K et al. (2013). "The Genomic HyperBrowser: an analysis web server for genome-scale data". *Nucleic acids research* 41.W1, W133–W141 (cit. on p. 28).
Schwartz, Barry (2004). *The paradox of choice*. Webpage. Ecco New York.
http://experiencelife.com/wp-content/uploads/2013/04/The-Paradox-of-

Choice.pdf
(cit. on p. 31).

Session history and navigation (2010). Webpage.
https://www.w3.org/TR/2011/WD-html5-20110113/history.html
(visited on 04/30/2016) (cit. on p. 54).

Simovski, Borris, Daniel Vodak, Sveinung Gundersen, Eivind Hovig, and Geir K Sandvik. "GSuite HyperBrowser: integrative analysis of dataset collections across the genome and epigenome" (cit. on p. 28).

Soeken, Mathias, Robert Wille, and Rolf Drechsler (2012). "Objects, Models, Components, Patterns: 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings". Ed. by Carlo A. Furia and Sebastian Nanz. Berlin, Heidelberg: Springer Berlin Heidelberg. Chap. Assisted Behavior Driven Development Using Natural Language Processing, pp. 269–287 (cit. on p. 35).

Stefanov, Stoyan (2010). *JavaScript patterns*. First Edition. Sebastopol, CA 95472: O'Reilly Media, Inc. (cit. on p. 15).

Syromiatnikov, Artem and Danny Weyns (2014). "A Journey through the Land of Model-View-Design Patterns". *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*. IEEE. Sydney, Australia, pp. 21–30 (cit. on p. 25).

URIs, URLs, and URNs: Clarifications and Recommendations 1.0 (2001). Webpage.
https://www.w3.org/TR/uri-clarification/
(visited on 03/24/2016) (cit. on p. 10).

Usage statistics and market share of JQuery for websites (2016). Webpage.
http://w3techs.com/technologies/details/js-jquery/all/all
(visited on 02/09/2016) (cit. on p. 16).

Yule, Daniel and Jamie Blustein (2013). "Design, User Experience, and Usability. Design Philosophy, Methods, and Tools: Second International Conference, DUXU 2013, Held as Part of HCI International 2013, Las Vegas, NV, USA, July 21-26, 2013, Proceedings, Part I". Ed. by Aaron Marcus. Berlin, Heidelberg: Springer Berlin Heidelberg. Chap. Of Hoverboards and Hypertext, pp. 162–170 (cit. on p. 12).

Zhang, Haimo and Shengdong Zhao (2011). "Measuring Web Page Revisitation in Tabbed Browsing". *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '11. Vancouver, BC, Canada: ACM, pp. 1831–1834 (cit. on pp. 2, 22).