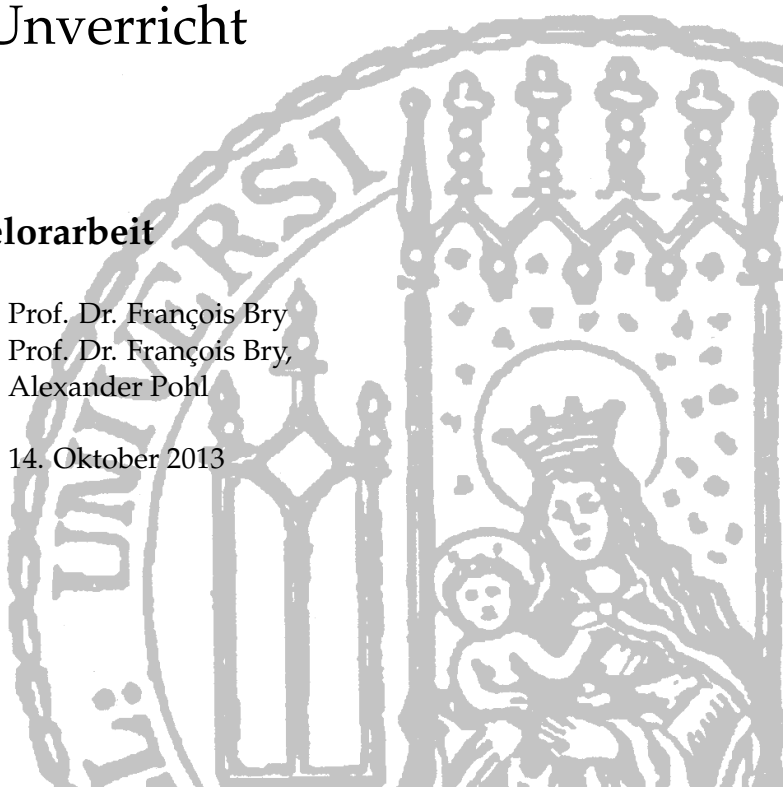
INSTITUT FÜR INFORMATIK
der Ludwig-Maximilians-Universität München

TAMING COMPLEXITY IN THE DEVELOPMENT OF JAVASCRIPT BASED RIAS

Case Study Backstage

Daniel Unverricht



Bachelorarbeit

Aufgabensteller	Prof. Dr. François Bry
Betreuer	Prof. Dr. François Bry, Alexander Pohl
Abgabe am	14. Oktober 2013

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe.

München, den 14. Oktober 2013 Daniel Unverricht

Abstract

This Bachelor's Thesis deals with a revision of the web client of Backstage, a backchannel for large class lectures. The goal has been to structure and to improve extensibility of the software, which has been developed in a rapid prototyping manner. Therefore, the principle "Separation of Concerns" is considered both as a technique in software engineering as well as a means to separate and simplify tasks in a development team. Recently improved JavaScript development tools and application frameworks (automatic dependency management, automatic testing) make Separation of Concerns in the development of the Backstage client possible. The Backstage client is structured using a MVC/P framework and the development process is revised such that it simplifies maintenance and extensibility of the client.

Zusammenfassung

Diese Bachelorarbeit beschäftigt sich mit der Überarbeitung des Webclients von Backstage, ein Backchannel für Massenvorlesungen. Das Ziel war, die bislang unstrukturierte und im Sinne des rapid prototyping erstellte Software zu strukturieren und deren Erweiterbarkeit zu verbessern. Hierfür wird das Grundprinzip "Trennung der Anliegen" (Separation of Concerns) sowohl als Technik in der Softwareentwicklung, als auch als Mittel für die Trennung und Vereinfachung der Aufgaben in einem Entwickler-Team erläutert. Eine solche Trennung der Anliegen ist in der Entwicklung des Webclients von Backstage erst durch kürzlich verbesserte JavaScript-Entwicklungswerkzeuge und -Anwendungsframeworks ermöglicht worden (z.B. automatische Verwaltung der verwendeten Programmbibliotheken und automatisches Testen). Der Backstage-Client wurde mithilfe eines MVC/P-Frameworks strukturiert und der Entwicklungsprozess so umgestaltet, dass nun eine Wartung und Erweiterung des Clients auf einfache Weise möglich ist.

Acknowledgements

I would like to thank Prof. Dr. François Bry for offering this interesting topic for my thesis and his advice.

Furthermore I owe gratitude to my supervisor Alexander Pohl with whom I had many a discussion about the specific subjects covered in this thesis and who always challenged my work in the most constructive manner.

Contents

CHAPTER 1

Introduction

This thesis discusses possibilities of a sustainable client-side software development of so called Rich Internet Applications (RIAs) using JavaScript, CSS and HTML. Until the year 2006 or so, the Web has mainly been a document retrieval system with limited interactivity. A user's request for new content on the server resulted in a complete exchange and rendering of the content in a browser. In essence (as it is considered nowadays), a web page consists of HTML that specifies the content and provides structure, CSS that specifies the layout (or "design") and JavaScript that adds dynamics and interactivity. But interactivity of web pages then had one significant limitation: All information had to be present as the page had been loaded. No further interaction with the server was possible (e.g. loading conditional content or having the server execute an operation on the database) without triggering another page-wide reload. As the Web's primary purpose has been looking up information, only a few interactive elements (e.g., dropdown menu etc.) were required. Ever since the year 2005, the major web browsers are equipped with the XMLHttpRequest object that enables clients to asynchronously, i.e., in the background, request data from the server and manipulate the currently rendered DOM, making entire page reloads redundant (cf. Wikipedia english, XMLHttpRequest). The XMLHttpRequest has become the basis for web applications that rely on Asynchronous JavaScript and XML (AJAX) and since 2006, the W3C has also been working on a standardization of the XMLHttpRequest (W3C XMLHttpRequest Working Draft [1]).

AJAX allows for a new level of interactivity, because it hides the static nature of web pages by performing necessary server communication in the background. A web page utilizing AJAX is often more than just a simple presentation of content: it

lets the user interact with that content. As that kind of interactivity resembles more a desktop application than a simple web page, the term User Interface (UI) can also be used to refer to the presentation of an interactive web page. Moreover the term web page does not sufficiently express the extent of interactivity. For that reason these web pages can be called RIAs ("can", because no meaning for RIA commonly agreed upon exists [2]).

The increase of interactivity entailed by AJAX has put higher demands on the dynamic manipulation of a web page through JavaScript. Besides these demands, various incompatibilities among browsers regarding accessing and manipulating a DOM have led to further difficulties JavaScript developers had to cope with. A significant simplification has been introduced by the jQuery[1] framework in the year 2006. jQuery simplifies interacting with the DOM of a HTML page by providing a rich JavaScript API. It adds another level of abstraction to the DOM interaction. jQuery aims at providing fast solutions to common scenarios when interacting with the client, e.g. dynamically changing text, adding info boxes or sending requests to the server. Statistics show the ever growing popularity of jQuery: 90% of web pages using a JavaScript framework use jQuery [3, 4].

AJAX has been unleashing highly interactive web applications that have shown to be in no way inferior to desktop applications. Excellent examples are Google Mail[2], which aims at replacing desktop-based e-mail clients, or Google Drive[3], an entire office suite similar to Microsoft Office or Open Office, which is available online and accessible via web browsers. As applications like these often offer a similar volume of functionality as their desktop counterparts, their development is at least as complex.

However, the development approaches of HTML/CSS/JavaScript applications being used until then have been stretched to their limits. Instead, the web client developer community now seem to also focus on development approaches that have been used to tame complexity in desktop applications. In the development of desktop applications the complexity is handled by Separation of Concerns, that is realized with architecture patterns such as Model View Controller (MVC) and Dependency Injection (DI), both of which are explained in Section 4.1, and by testing the software (see Section 2.2.2). Many new frameworks that implement or incorporate these approaches have appeared in the landscape of web client development. Each of them provides support for different development approaches: while some provide an MVC architecture, others also aim at simplifying object-oriented development in JavaScript. The framework also may have a focus on a certain kind of application such as a web client for mobile devices.

Backstage is a collaboration platform designed to support and sustain interaction of learners in large, i.e. 80 students or more, lectures [5]. For this purpose, Backstage provides short messages (microblogging) as a means to collaboratively

[1] `http://www.jquery.com`
[2] `http://mail.google.com`
[3] `http://drive.google.com`

annotate slides using predefined message types. Besides communication Backstage realizes functionalities of Audience Response Systems, viz. quizzes during lectures which are answered by students on Backstage and which results are summarized and displayed in real-time. The range of functionalities offered in Backstage requires the support of partly complex interactions among students and between the audience and the lecturer. The complexity of Backstage's client side has reached a degree of complexity that seems to warrant the use of a framework in order to simplify maintenance and further development. This thesis aims at finding out if simplified maintenance and better extendability is possible using a framework and tools available today.

The outline of this thesis is as follows: Chapter 2 discusses different aspects that have to be taken into account when deciding on a framework. In Chapter 3 the different components which make up a RIA are analyzed. Based on this analysis it is explained how one can benefit from the concept of Separation of Concerns by separating these components into separate logical units that communicate via defined interfaces. The new implementation uses the framework AngularJS[4], that is introduced in Chapter 4. Chapter 5 discusses the current implementation of the Backstage client, highlights its problems and explains how the concepts and methods established earlier can be used for a more structured web client architecture. Additionally Chapter 5 discusses how the refactored architecture of the Backstage client facilitates maintenance and aids further development.

[4] `http://angularjs.org`

CHAPTER 2

On Frameworks for Web Clients

This chapter discusses criteria that should determine the choice of a framework. It demonstrates how important differences can be discovered and how these differences can influence the decision in favor of one or another framework.

2.1 Types of Frameworks

Frameworks can be distinguished according to the abstraction and the programming model they provide. Frameworks reduce complexity by providing a standardized way of implementing an application. However, an application's architecture has to comply with the framework's philosophy: a large deviation from the intended use of a framework, e.g. the use of a NoSQL-database with a framework that only supports relational databases, can yield many difficulties which even may end up in "fighting against the framework". Additionally, frameworks differ in the extent to which they structure an application. At the low end of the range are frameworks that only supply a JavaScript object for each of Model, View and Controller and hide the logic that connects these three objects (e.g., Backbone JS[1]). Others aim at presenting mainly textual content for mobile devices and therefore provide only mobile friendly UI elements and navigation concept (e.g., jQuery mobile[2]). In most cases, frameworks have distinct purposes, even if some of these purposes are just providing as much flexibility as possible or forcing the developer to incorporate certain architecture patterns. One may summarize that the more a

[1] http://backbonejs.org/
[2] http://jquerymobile.com/

framework focuses on a specific type of application, the less effort is required for the development of applications of this type.

2.2 Criteria for Choosing a Framework

2.2.1 Conformity With the Application to be Developed

The choice of a suitable framework makes a close examination of the requirements at hand necessary. For example, a project could depend on the use of jQuery-UI for calendar, scrolling or any other features. In that case the framework must be compatible with any of jQuery's implementations and allow its use within the framework-code. Another compatibility requirement is the inclusion of "old code" in the new project, the ability to include working functionality into the new application structure without a major revision (Frameworks like AngularJS provide support by encapsulating any working JavaScript code in a so-called directive exposing its endpoints in a Angular-ready API).

2.2.2 Testing

More complex applications require additional steps to ensure code quality. Software testing has become a vital part in the development of desktop applications. Recently new technologies have been made available that enable the JavaScript developer to test web client code as well. Especially dynamically typed scripting languages such as JavaScript require thorough testing, since these languages are interpreted and, thus, rather simple errors, e.g., type errors or syntactic errors, cannot be recognized as easily as in statically typed languages. Therefore the framework needs to support or even facilitate testing the software using unit-tests, end-to-end-tests or integration-tests (see Section 5.3.2).

2.2.3 Integration

The client-side development should be compatible with existing source code management (SCM) processes (e.g., automatic testing, deployment). Additionally, the toolchain and the framework should support tasks such as minification of code, that is compression of files by removing unnecessary whitespaces and renaming variables and function names in order to reduce network transmission costs of HTML, JavaScript and CSS, which are specific to the development of the client-side.

2.2.4 Documentation and Community

It is desirable to use a successful framework, as a successful framework is well maintained and documented. In turn it is also possible to say, that the success of a framework depends on the quality of its documentation. The less one is able to understand a framework by studying its documentation, the more effort is required in acquiring the skills to use it (e.g., a developer, who joins a project, needs to get acquainted with the new technology, so a well-written documentation is not a once in a project-lifetime necessity but may decrease recurring training costs). Hence the quality of the documentation allows to estimate how successful a framework is likely to become. Currently the two leading sources of information about a new framework are its online documentation and forums like stackoverflow[3]. Another factor is the community that is already using the technology. Are there active forums that help developers in their tasks? If so, are there any indicators about its size and activity? Again, stackoverflow can be used to take an estimate about a framework's community size and activity. At the time, simple tag counts at stackoverflow produced the following results (Figures 2.1 and 2.2). This data allows a cautious analysis on the community behind these frameworks.

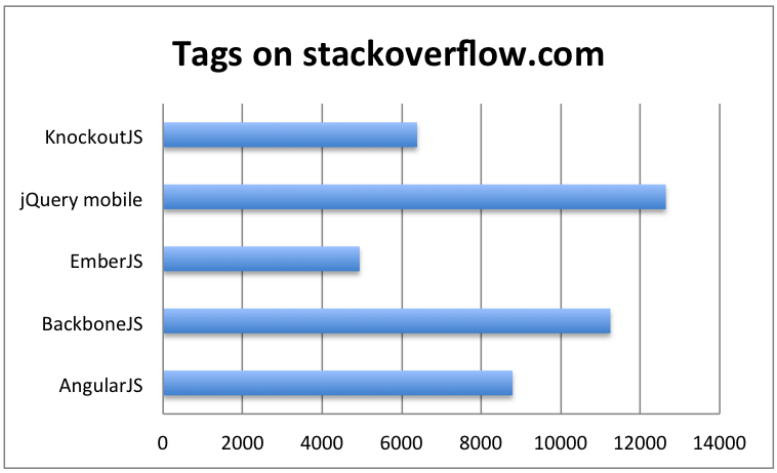


Figure 2.1: How many posts are tagged with a framework (total)

[3] `http://stackoverflow.com/` is currently the leading forum for developers. For example, in Germany applicants for IT jobs are especially invited to provide their stackoverflow profiles.

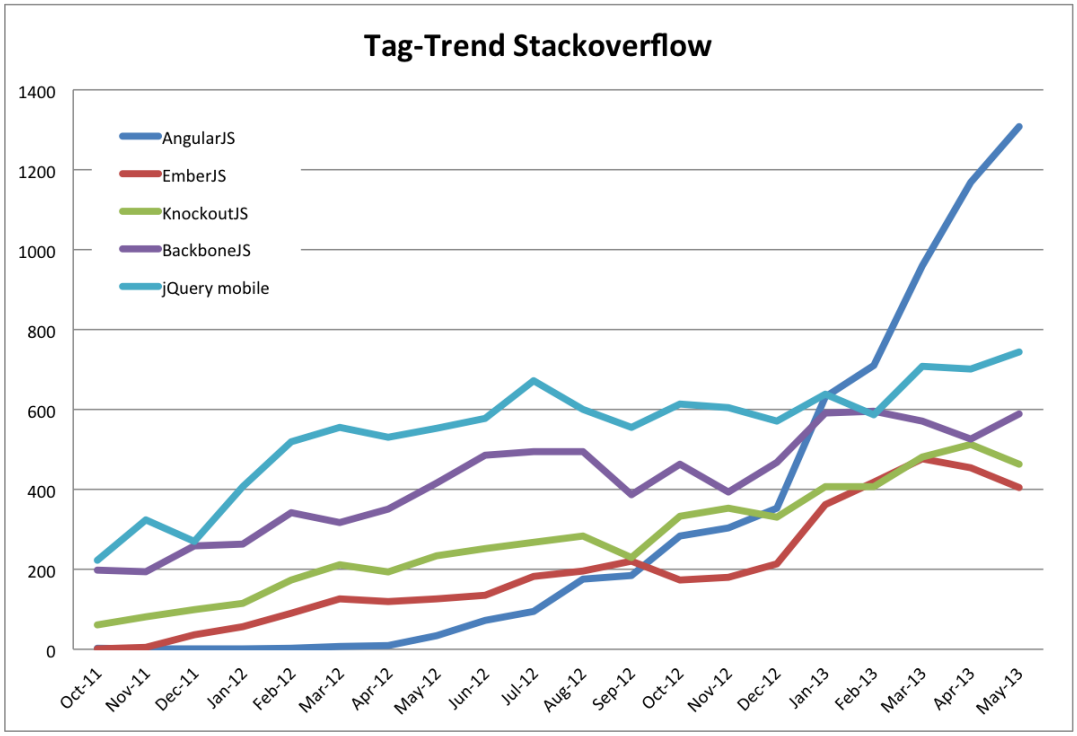


Figure 2.2: Number of posts tagged with one of the five most popular framework names (analysis over time). The more posts are tagged with a framework, the more people are interested in this framework.

2.3 Sustainability

The longer a framework is supported by its vendors or developers and a thriving community, the more likely the applications remain sustainable. Especially in the field of frameworks, development is so fast that current frameworks are frequently outstripped by new developments. Thus, programming languages and frameworks alike have an unclear durability. Large companies backing the product or a large and active community, however, can be indicators of sustainability.

2.4 Frameworks' Intentions and Complexity in its use

The objective of a framework is to standardize software architecture so as to simplify development. A particular problem could be solved in two lines of code using a suitable framework or result in an unmanageable mass of code using an unsuitable one. Usually a framework developer has his ideas and beliefs about best-practices or the best programming style. Of course, these opinions are reflected in the product, the particular framework. If one is about to use a framework, the ma-

jority of the developer team should agree with its "opinions". Besides opinions one also has to consider the learning curve and thus training costs. Sometimes, a framework is built upon further abstraction provided by other frameworks that might increase the complexity of the software. For example, BatmanJS[4], although its name suggests otherwise, requires CoffeeScript[5], a meta-scripting-language which needs to be translated into JavaScript.

2.5 Developers' Preferences

Similarly to choosing the programming language the choice of a framework is also a matter of taste. Different frameworks approach web client development in a different fashion. For instance, the Web as we know it relies on HTML as the view, the use of which, however, is not appreciated by every client developer. JavaScript enhances the concept of traditional static HTML elements whose presentation is controlled by CSS. Two different strategies become obvious. Developers, who do not like the idea of HTML as a UI language, prefer to reduce HTML pages to only contain a basic layout, that then is modified and managed by JavaScript. Other developers, in contrast, stress the declarative nature of HTML by augmenting it with additional tags and attributes. The latter approach results in a more declarative "language" and the former in a less declarative one. As mentioned above, a general recommendation (i.e. one framework satisfying every developer) is not possible, because it does not make sense to argue about personal taste. If web designers are involved, the use of HTML as the UI language for the web client might be a prerequisite.

[4] http://batmanjs.org/
[5] http://coffeescript.org/

CHAPTER 3

Architecture

3.1 Separation of Concerns (SoC)

A well-known paradigm for maintainable software is Separation of Concerns [6] (SoC). SoC represents the idea of splitting an application into components, that each is concerned with a distinct task. These components are largely independent from each other and communicate via a well-defined interface. A prominent example of SoC is the layered design of the ISO/OSI [7] stack, where every layer can be exchanged easily, as long as the defined interfaces to adjacent layers are respected. In RIAs, SoC can be achieved in multiple ways: in defining layers that make up the application and in identifying and separating tasks within these layers (e.g. objects and their task-specific relationships). There are different groups of people involved in RIA development (e.g., web designers, JavaScript developers, etc.). SoC can also be used enable each of these groups to operate as independently as possible. The following sections discuss the applications outlined above.

3.1.1 SoC: Layered Architecture

In RIA clients SoC is realized by implementing layers. A RIA server makes application data available to the RIA client by the means of a well-defined interface, e.g., by providing Unique Resource Identifiers that deliver data in some format upon being called. That is the interface the client has to implement. Possible choices for layers are the following.

A *Business Layer* of the client provides a single point of access to the backend, processes user interaction and makes data available for the *Presentation Layer*. The Presentation Layer presents the UI to the user. Each of these layers can be extended, modified or exchanged as long the the interfaces are respected, thus leading to maintainable software. The following section goes into detail about how SoC is used to further structure the Presentation Layer

3.1.2 SoC: Within the Presentation Layer

SoC within the Presentation Layer is achieved by using architecture patterns such as DI and MVC (see Section 4.1). Many available frameworks help in that respect. They define where to put View-code, Controller-code and Model-code and hide their interactions. In different words, developers are able to transfer much of the resulting complexity into the framework. (Figure 3.1)

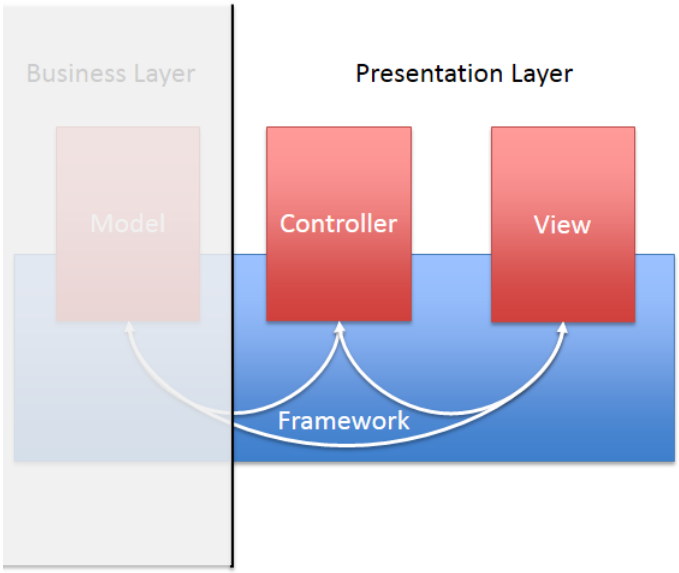


Figure 3.1: The framework takes care of components' "wiring".

3.1.3 SoC: Separating the Developers' and Designers' Concerns

Separation of Concerns is not only a paradigm valuable in software design but also for separating the concerns of a development team's members. For example, a web designer should not have to install the server application and a database server in order to develop the client-side. Taking one more step towards the separation of front- and backend development, the two do not only have to be separated physically (on different machines), but also on a conceptional level. Defining an Application Programming Interface (API) between the two achieves that. In web devel-

opment that kind of API consists of the definition of URL(s) in conjunction with a fixed data format. That way the backend developer does not need to bother with frontend related matters. In turn the frontend developer only needs to know the backend-API thus eliminating setup complications. *Implementation*. As many web frameworks do not support the desired separation of front- and backend development, it has to be accomplished another way. There are two requirements: firstly, the backend needs to be run on a different physical machine but still be accessible by every developer. There are many ways to achieve this, among these is the use of a Continuous Integration server (see Section 5.3.3) on which the current version of the backend is deployed for everyone to access. Additionally, a security measure of many modern browsers, the Same-Origin-Policy [8], which prohibits AJAX-requests across multiple domains (as is the case if the server application does not run on the same machine as the client), has to be circumvented. A server, which delivers all documents pertinent to the client-side from the local machine and proxies all requests to the backend to the remote server, can lead browsers to believe that all data originates from the same domain (see Section 5.3.1 for more detail), thus solving the problem. Secondly, the backend-communication needs to be defined a priori resulting in a clear understanding for both programming parties of how front- and backend will interact. With a setup like this no client developer needs to install a possibly complex backend on his machine.

Furthermore JavaScript developers and web designers have different concerns as well. A web designer wants to be able to create and change the design of the application without having to bother with the client side application logic (i.e. he does not want to or is simply unable to concern himself with the inner workings of JavaScript code. He should be able to easily look at the HTML structure, change minor parts of it and implement CSS and graphics accordingly). As mentioned in Chapter 1, early web sites had little interaction possibilities and therefore contained little and simple JavaScript code which did not present a problem to a web designer in that respect. In AJAX-based applications, design is but some small part in the whole application. Chapter 4 explains how the framework AngularJS presents a possibility for decoupling design from logic in the frontend.

CHAPTER 4

Introducing AngularJS

AngularJS is a JavaScript-based framework that aims at supporting the development of large and complex clients. As opposed to other client frameworks such as Sencha [9], which almost completely abstracts from HTML in favor of JavaScript, AngularJS stresses the advantages of HTML as a declarative language that both web designers and JavaScript developers understand and can work with. It does so by connecting JavaScript application logic with the View through special attributes added to HTML elements (see Section 4.2.1).

4.1 Preliminaries

AngularJS realizes SoC in the presentation layer of a RIA client by implementing the architecture patterns Model View Presenter (MVP) and Dependency Injection (DI), both of which are briefly described in the following section.

4.1.1 Model View Controller (MVC) and its successor Model View Presenter (MVP)

A succinct and to the point definition of MVC, a forerunner to Supervising Controller / Passive View, is given in [10]: *"MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input. [...] MVC decouples views and model by establishing a subscribe/notify protocol between them. A view must ensure that*

its appearance reflects the state of the model. Whenever the model's data changes, the model notifies views that depend on it. In response, each view gets an opportunity to update itself. This approach lets you attach multiple views to a model to provide different presentations. You can also create new views for a model without rewriting it."

One can find several variations of the MVC-pattern. The main concept behind them is the separation of Model, View and Controller. They can be merely distinguished by the principles that dictate how Model, View and Controller are synchronized and the location where application logic is performed. (Figure 4.1)

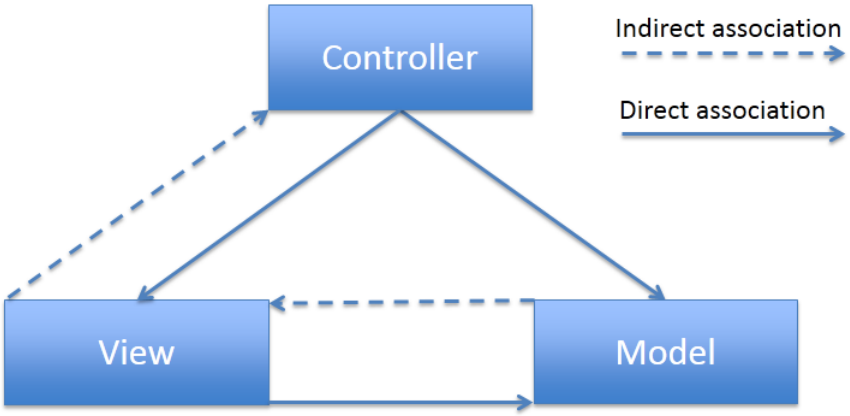


Figure 4.1: Classic MVC. A indirect association is for example implemented using the Observer pattern [10], a direct association between two objects A and B exists, if A holds a reference to B or vice versa (Figure adapted from [11]).

A variation of that pattern is Model View Presenter (MVP). It allows communication between View and Model in both directions and additional View-logic may be located in the Controller. Martin Fowler further describes a pattern called Supervising Controller / Passive View [12] that evolved from, or more accurately explains, the MVP pattern, where the Presenter assumes additional responsibilities which were originally the View's. In the case of RIA clients built using AngularJS (see the following sections), Fowler's variation of the pattern applies, because the HTML-document acts as the View and is thus unable to perform active tasks due to the passive and declarative nature of HTML. If Model-variables and View-variables are continuously kept in sync, the concept of *data-binding* [13] is applied.

4.1.2 Dependency Injection

Dependency Injection (DI) is the method of moving the creation of objects, application components depend on, from the application into the framework, i.e. the application itself does not need to be aware of concrete implementations any longer.

The need of Dependency Injection can be seen as a consequence of "Design-by-Contract"-software architectures, according to which components are implemented against interfaces ("contracts") rather than concrete classes. Components that serve other components in accomplishing their tasks are also referred to as collaborators in this context. Implementing against interfaces provides for a clear separation between specification and implementation and allows for an easy replacement and maintenance of collaborators on the basis of contracts, aspects that gain in importance as software grows. Yet, at some point, collaborators need to be instantiated, i.e. constructed from concrete classes. The goal of Dependency Injection is to have the framework create the collaborators and to "wire" them together, i.e. "inject" the created collaborators as references into components, according to a (usually declarative) specification provided by the developers. As the control of collaborator creation and dependency resolution no longer lies with the application but with the framework, Dependency Injection is also considered as an instance of Inversion of Control (IoC, [14]) and the part of the framework performing Dependency Injection (among other things) as the IoC Container. Although this section does not go into detail any further it should be noted that IoC entails significant changes in the approach of software design as numerous parts such as start up, shut down, initialization and configuration, but also execution of business logic is now triggered by the IoC container, thus pushing the application into a rather passive role. This is why IoC is often associated with the Hollywood principle "Don't call us, we call you" [15]. In statically typed languages collaborators can be, and are often, injected "by type", meaning that collaborators are created and injected according to the interfaces they implement. The injection can either take place on construction of a component, i.e. on calling a component's constructor (constructor-based injection), or immediately after a component has been constructed (setter-based injection) [14]. The latter seems to be more commonly used, possibly as it allows for an easier resolution of cyclic dependencies among components and collaborators. Injection may also be accomplished "by name", meaning that collaborators are injected whose names match the reference names specified in a component. This kind of Dependency Injection is of particular significance to dynamically typed languages like JavaScript and is also used in AngularJS as described in the text below.

4.2 Angular in a nutshell

4.2.1 Concepts

Although the documentation states that AngularJS is based on MVC, a closer look reveals that it is rather based on MVP or as described more recently in [12], on Supervising Controller/Passive View. The synchronization of the View and the Model is accomplished by data-binding (see Section 4.1.1). The HTML[1]-code acts as the Passive View. By means of an attribute, any HTML node can be connected

[1] AngularJS is compatible to any HTML dialect available

with a controller. A controller is associated with a scope, a variable storage that holds variables that should be synchronized by data-binding and that should be available in the HTML template. In the development of larger software it is common to separate controller logic and business logic, i.e. the logic not concerned with MVC-related tasks, by encapsulating the business logic in so-called services. Controllers reference the services and call the business logic to be performed. In most cases, single instances of services are sufficient. Hence, services are often realized using the singleton pattern [10]. Being singletons, services should also be stateless and can be uniquely referenced by name, which also makes DI by name possible. AngularJS appreciates the declarativity of HTML as a UI language. However, HTML is passive in the sense that it cannot execute application logic (but define action handlers that, however, need to be provided in JavaScript). AngularJS gives the developer the ability to create custom tags in the HTML structure that encapsulate another set of a Model, a View and a Controller. These custom tags can communicate by using the same service to load and store data.

4.2.2 Lifecycle

All the complexity in AngularJS boils down to two mechanisms: Dependency Injection and data-binding. But how can these two be achieved using JavaScript? AngularJS implements them in two phases. One is called *compile-phase* the other *link-phase*. With the help of a simple application (see Listing 4.1) AngularJS' approach is analyzed.

```
1  ...
<body ng-app>
3    <div ng-controller="exampleController">
       Regular Text {{variableName}}
5    </div>
     <script type="text/javascript">
7      function exampleController($scope, myService) {
         $scope.variableName = "Hello World";
9      }
       angular.module('ModuleName').factory('serviceA', [function() {...}];)
       ;
11   </script>
</body>
13 ...
```

Listing 4.1: A simple AngularJS application

4.2.2.1 The Compile-Phase

After loading the HTML page (with all necessary scripts and stylesheets) the compile-phase begins. The DOM is traversed saving all node names and their attributes. Additionally all node-content in `{{}}` (see line 4 in Listing 4.1) is parsed.

As JavaScript supports exporting all function- and variablenames as strings, another list is created containing these. In the sample application AngularJS' compile-phase would proceed as follows

- Begin DOM traversal of the HTML page
- Found `ng-app` attribute: AngularJS is made aware that this page contains an AngularJS application. Every child node is designated as part of it.
- Found `ng-controller` attribute: Save value and DOM-node for further processing in the link phase
- Found `variableName` within `{{}}`: Save `variableName` with containing node for further processing in the link phase
- Begin reading JavaScript code and saving all function signatures
- Save `exampleController`-reference in the code for further processing in the link phase
- Note, that `exampleController` needs `$scope` and `myService` for further processing in the link phase
- Save `serviceA`-reference in the code for further processing in the link phase

4.2.2.2 The Link-Phase

As JavaScript is a dynamically-typed language, Dependency Injection by name is used. The AngularJS engine needs to use variablenames in order to find the right object to inject. In the link-phase all function names are matched against the parameter names on a string comparison basis and then, in case of a match, instantiation will take place (Figure 4.2).

- Found a match: `serviceA` matches parameter in `exampleController`
- Instantiate `serviceA`
- Instantiate `exampleController` by creating a `$scope` object first ($scope is a special argument that is automatically instantiated by AngularJS, it will be explained further in the rest of this section) and then invoking `exampleController()` with references to `serviceA` and the `$scope` object created earlier

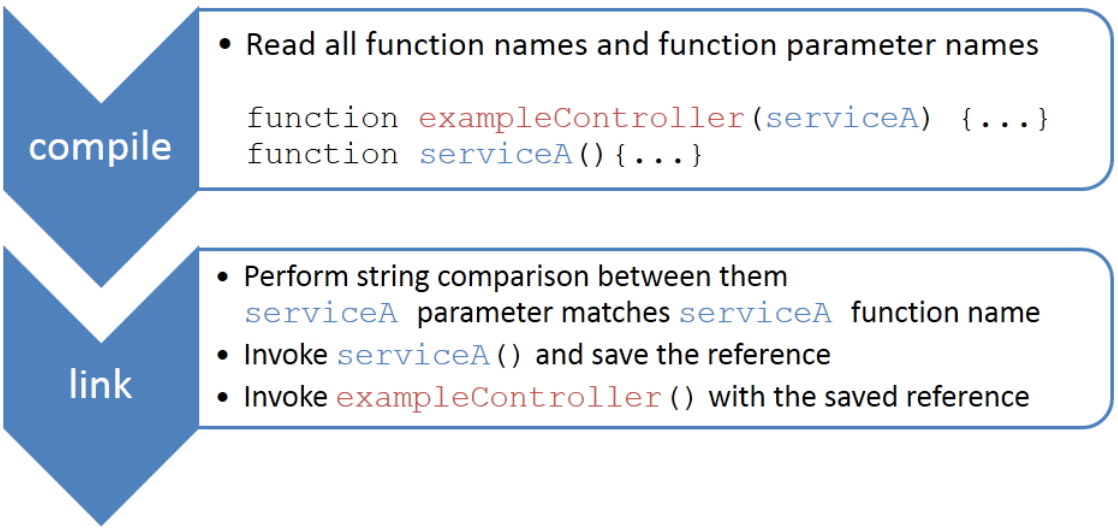


Figure 4.2: Accomplishing Dependency Injection by name

4.2.2.3 Data-Binding

Subsequently the View (HTML) is connected to the Model via $scope and data-binding. Every Controller is connected to a DOM node by evaluating the `ng-controller`-attribute specifying the underlying Controller name. Each controller is associated with a scope, addressed by `$scope`, that holds all data-binding variables and AngularJS expressions that require evaluation by the controller. In Listing 4.1 `$scope` would contain `variableName` and be initialized with "Hello World" as specified in line 8. AngularJS connects these scope variables to their DOM counterparts via data-binding (Figure 4.3). In order to avoid inconsistencies AngularJS implements a loop that compares variable values with their values in the previous cycle (dirty-checking) and fires a custom change event if they do not match [16]. This event, in turn, is caught by a listener within AngularJS updating the corresponding outdated variable. AngularJS calls this loop `$digest`. Using dirty-checking as opposed to listening to the standard change events distinguishes AngularJS from other frameworks like KnockoutJS oder BackboneJS. This is because dirty-checking can be implemented using only the parts of JavaScript which are consistent across modern browsers, and the framework has to be able to decide if and when to initiate the synchronization process, whereas standard change events always fire when a variable changes. In a scenario where two arrays have to be kept synchronized every single change will fire an event on one side, be changed on the other side firing another change event. This complexity has to be managed (see [17] for a more detailed explanation). AngularJS's developers assume, that a cycle checking every single variable for changes, is still fast enough for human perception [17]. The `$digest`-cycle can also be turned on externally, which allows

the use of external code not included in the AngularJS application.

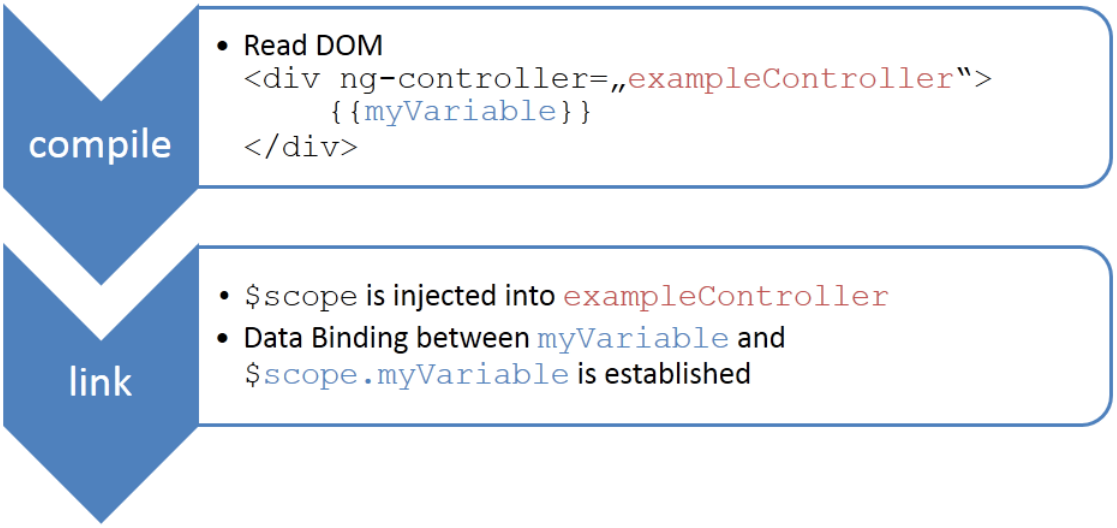


Figure 4.3: Accomplishing data-binding

4.3 Why AngularJS

To evaluate which of the three frameworks in our short list, AngularJS, EmberJS and BackboneJS, should be used to reimplement the Backstage-client, a simple login procedure was implemented. Afterwards the resulting code for this little project was examined. A clear favorite emerged, as, once understood, the code for the AngularJS framework forced the developer to use a certain structure for the code, which led to well-arranged code (The liberties other frameworks provide in that respect did not enforce a standard). AngularJS relies on enriching existing HTML with custom elements and attributes which leads to very declarative HTML-code. These custom elements implement another even smaller MVC-pattern as they connect a HTML-Template to a JavaScript-Controller which in turn accesses the global Model of the application.

An exhaustive documentation made it clear that the main pattern behind AngularJS is Dependency Injection which facilitates testing and modular software-architecture. It provides mechanisms for incorporating legacy-code using other frameworks by encapsulating it within one of these custom elements. As these elements and their functionality are connected by DI, a slow transition from other frameworks to AngularJS is as easy as excluding the old class and including the new one. (Figure 4.4).

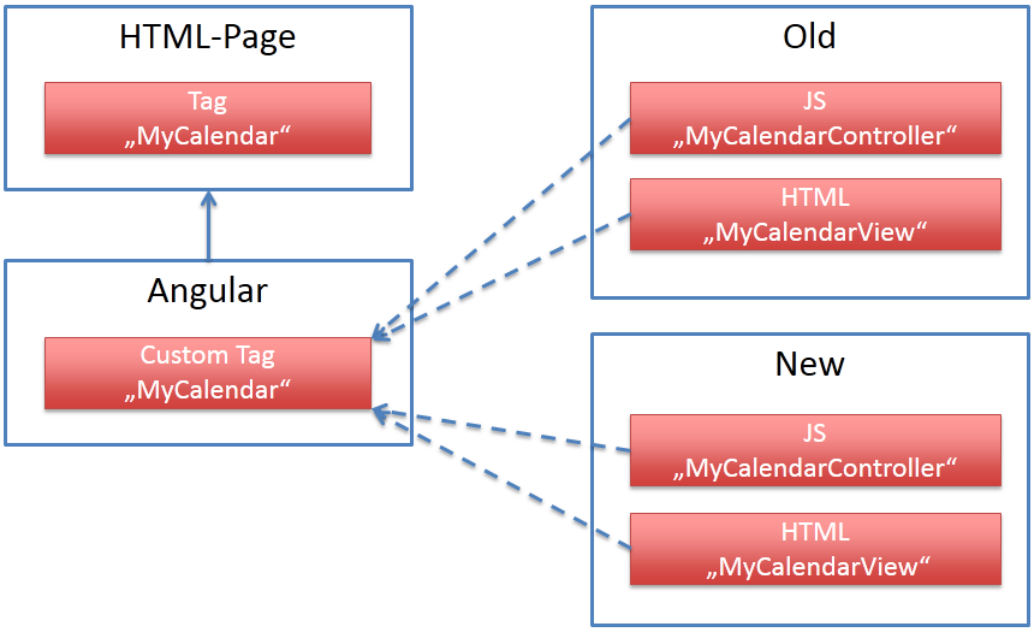


Figure 4.4: Directives connect DOM-nodes with a template (View) and JavaScript-logic (Controller)

Features like data-binding (4.2.2) eliminate boilerplate-code. An active community on the documentation pages and on stackoverflow present a simple means of acquiring knowledge about the framework. The effort Google has already been investing in the AngularJS could be an indicator that the framework is going to be around for longer.

CHAPTER 5

Backstage - A new Approach

5.1 Current Client

A reimplementation of the Backstage client is necessary, since the client has been developed with jQuery which, as mentioned above, provides few means to structure an application. Since Backstage has been developed in a rapid-prototyping manner, Separation of Concerns has not been rigorously realized. The Backstage client has been developed with the purposes to quickly become ready-to-use in experiments. Thus the client lacks structure and extensibility. Furthermore, the client contains several pieces of "dead code", i.e. code that is unused. The Backstage frontend is split into fifteen modules that communicate by means of a simple event bus called EventHub. Whenever the client receives some kind of event from the user or the server, this event is published in the EventHub (Figure 5.1). Every module that is affected picks it up, processes and modifies event-data and republishes the event for other modules to do likewise (Figure 5.2). As multiple modules are working with the same event, problems can arise in terms of module-order in the EventHub. The application-state is distributed among all the fifteen modules, so whenever one is trying to extend the client or implements new functionality, the necessary parameters could be in different modules only at one specific step of an event workflow. Modifying the event at the wrong step or republishing it with the wrong values could mean breaking the whole application. In order to ensure this does not happen, every possible permutation of events needs to be tested manually before the new feature can be released. That procedure requires more effort each time a new module is connected. Additionally not all modules are connected to the EventHub yet and use different and inconsistent ways of achieving their purpose.

Thus the original EventHub also affects the extensibility of the client.

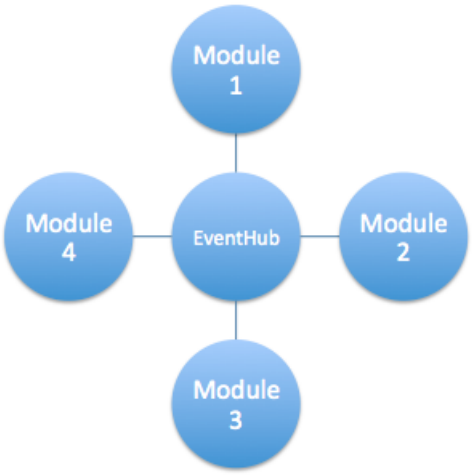


Figure 5.1: The original Backstage EventHub

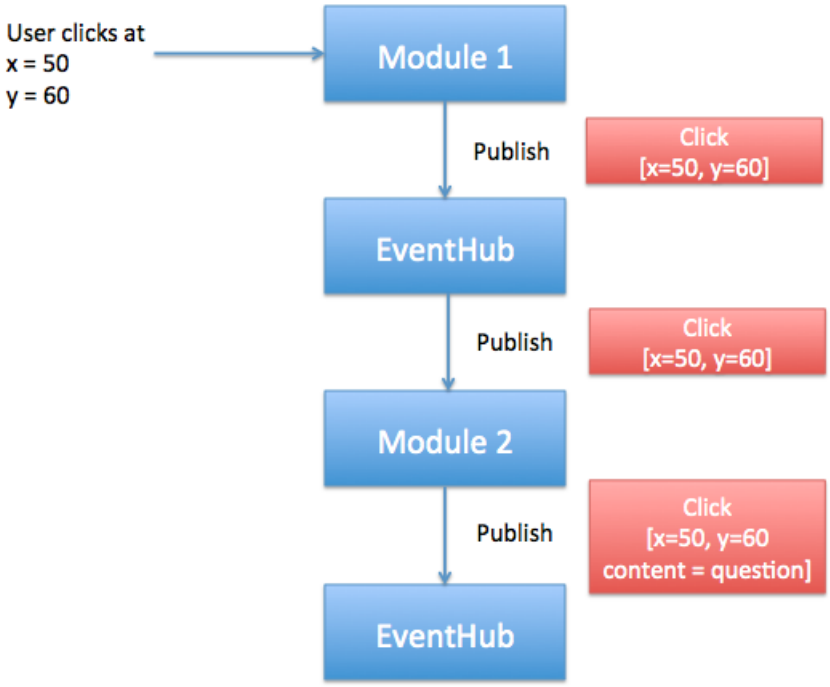


Figure 5.2: An illustration of the event flow

Content, that is needed after rendering the first page, is loaded in the background using AJAX-calls. Where possible, responsibilities have been defined for every module. Whichever module first needs additional data from the server

makes the AJAX-call updates "its" View-element and then propagates the data. The way this View-update works is as follows: A HTML-Page consists of HTML-elements which can be found using selectors. Once an element is found its attributes and content can be manipulated thus storing application-state in the element. For example, in a worst-case scenario where a View-modification depends on other elements being displayed, all elements involved have to be found, changed and new events triggered. In the case of the implementation of a new feature, a simple DOM-modification of the template could break this fragile concept.

By design this strategy saves the application-state in various places at once. RIAs may already feel like Desktop-Applications but the browser as medium of displaying remains. With it come possibilities of interaction the programmer cannot preclude: every user is able to navigate backwards and forwards in the browser history or reload the current page. The application needs to be consistent across all these interactions. Currently there is little to no plan in place that accommodates this requirement.

There are many different JavaScript libraries available that are supposed to help with many different problems. These libraries use different APIs. Changing the behavior of a module using an additional library involves further study of this library. Backstage currently uses at least seven that are still under development. Usage of these libraries demands timely updates and proper maintenance if for example the API changes. Prominent examples of such libraries are Direct Web Remoting[1] (DWR) that allows for real-time-communication between the RIA and the backend or KineticJS[2] which facilitates manipulation of the HTML5 Canvas-element

5.2 Current Server

The server-side of Backstage depends upon the Grails-framework for delivering its content. In a web application, the server-side could be described as Model and Controller, the HTML and JavaScript part (client-side) as the View in a MVC-pattern. One can argue, that with Grails it is not possible to talk about server- and client-side supported by the fact, that Grails abstracts from the client so that it only exists implicitly in server code. While generating HTML every backend variable is accessible and can be used for dynamic content. The server will render different pages according to state, role and request-URI. For smaller applications this all-in-one approach as a benefit: one technology, one API for the entire application. But as the programs grow larger, the complexity, that comes with it, becomes increasingly harder to handle. For changes in "backend-logic" the HTML-templates need to be adapted. Changing HTML-templates in highly interactive web applications almost always involves adapting JavaScript as well. Therefore a little change triggers

[1] http://directwebremoting.org/
[2] http://kineticjs.com/

a whole cascade of changes throughout the software. An experienced developer may wish for defined interfaces between the different components of client-side MVC.

5.3 Toolchain

5.3.1 Separating Client and Server

First of all, the overall complexity of the development setup has to be handled. One possible solution is decoupling the client architecture from the server entirely. Their only means of communication is specified in an Application Programming Interface (API). The client is able to switch from server to server by the means of changing one string in the code.

Implementation. The new implementation is based on a hybrid concept of Representational State Transfer (REST) and Remote Procedure Calls (RPCs). Unique Resource Identifiers (URIs) paired with a data specification define the only interface between client and server. Instead of loading full HTML-templates via AJAX call, the only possible resources being loaded are data in JavaScript Object Notation (JSON-data). This step effectively extracts any View-elements from the server, degrading it to a pure data-service. As a very welcome side-effect any client developer only needs a Unique Resource Identifier (URI) to an existing server and can develop against that without worrying about any dependencies the server might have. But a server that delivers HTML-content is still needed.

NodeJS[3] is a JavaScript runtime environment outside of the browser. NodeJS is most often used as a runtime environment for JavaScript-based implementation of the server-side of web applications, e.g., as part of the testing toolchain or for rapid prototyping. The task runner Grunt (similar to Apache Ant oder Unix' make, but based on NodeJS) comes with several modules, each performing a specific task for the developer. Among these is a web server that can deliver HTML-content under a certain URI. Since all client-side code is written in HMTL/JavaScript Grunt is well-suited. Implementing a conditional proxy in Grunt makes it possible to decide which URIs will be answered locally and which remotely (Figure 5.3) while solving the Same-Origin-Policy-problem mentioned in Section 3.1.3.

[3] `http://nodejs.org/`

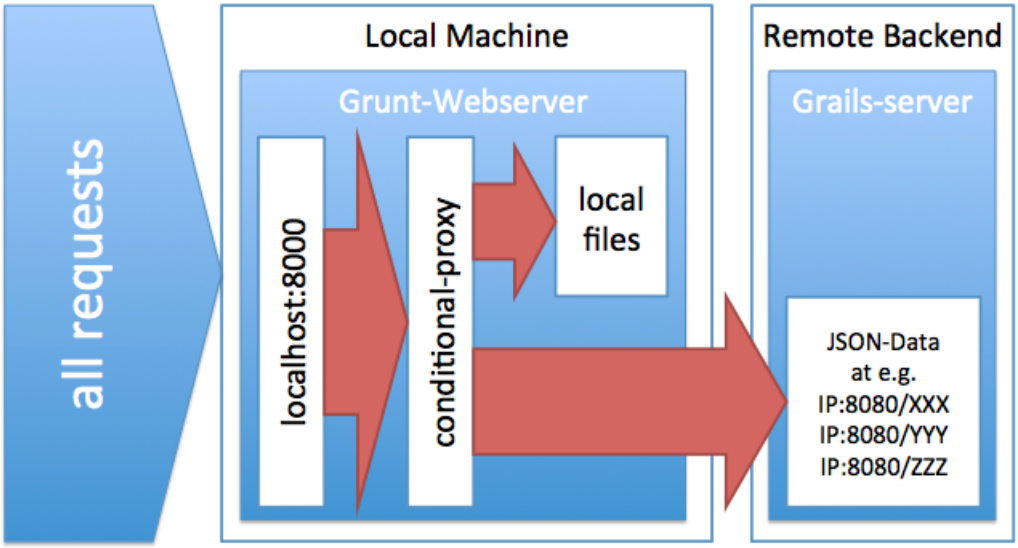


Figure 5.3: How frontend and backend are separated

5.3.2 Structured Tests

One way of increasing stability and therefore quality of code is by testing it. Unit-tests are written to ensure that small parts of code or logic do exactly what they were designed for. End-to-end tests imitate user interactions and evaluate their effects. When writing unit-tests for software, the programmer is made aware of all the dependencies a class or group of classes has. However, testing is only effective if the tests can be executed with minimal effort and are easy to set up. Development environments that support automatic testing execute tests automatically on events such as saving edited source files.

Implementation. Due to the architecture of AngularJS every injected service or object can be identified easily by looking at the function signature. If all global variables are decoupled from the DOM as described above, it becomes very easy to find out which objects have to be mocked and which changes to the model have to occur, thus facilitating the creation of unit-tests. Furthermore, AngularJS contains built-in support for end-to-end-tests. Using the same toolchain as discussed in Section 5.3.1, an additional task is created using Grunt. This task executes all tests located on files in a certain folder. These tests are written using Jasmine[4] and the AngularJS test-API. If that task is running and a developer as much as saves a file, all the tests will be executed within a few seconds and results displayed.

[4] `http://pivotal.github.io/jasmine/`

5.3.3 Continuous Integration

As explained in Section 2.2.3, complex RIA clients need to be assembled before deployment. The concept of Continuous Integration (CI) describes the automatic deployment of the newest version of a web application on a server, execution of all tests on the newest version of the application and the notification of the development team in case the tests are not passed successfully.

Implementation. A Grunt instance on the CI server can be used to build the RIA client, run all written tests and deploy it on a server.

5.4 Concepts of the new Implementation

5.4.1 Avoiding Explicit Dependencies Between Client-side Code and DOM Elements

In order to be able to easily add and remove functionality and components from and to the application, all possible dependencies to the DOM-structure have to be removed. A critical scenario could be a class accessing a variable from its parent (for example via DOM-selector). After moving this class to another point in the DOM the application could break. A central model with a clearly defined interface storing all data is a possible solution. (Figure 5.4)

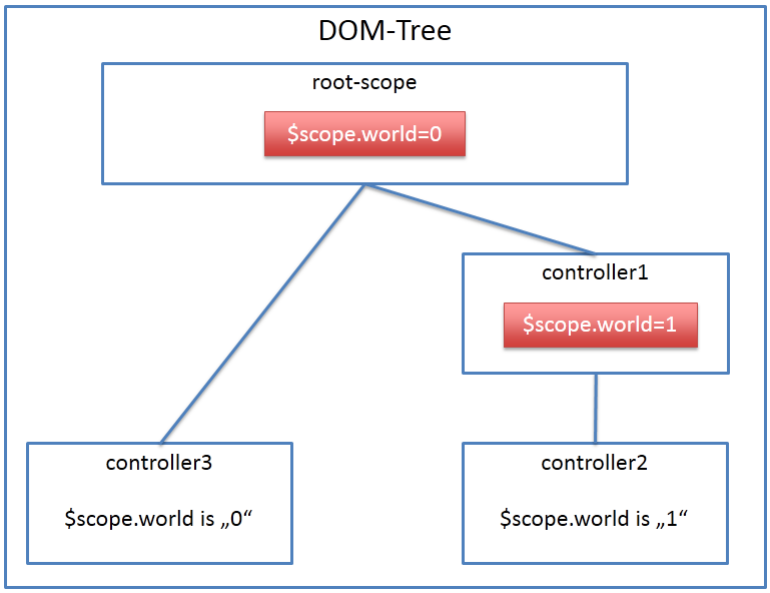


Figure 5.4: Depending on the DOM-structure world has a different value

Implementation. In this implementation a slight deviation from the AngularJS-standard (see 4.2.1) makes it possible to define an interface for communicating with the model. The root-scope will not be used but instead a service injected via DI. That way the programmer always knows which variable is accessed (no matter the function-scope) and can be sure it will always be accessible. (Figure 5.5)

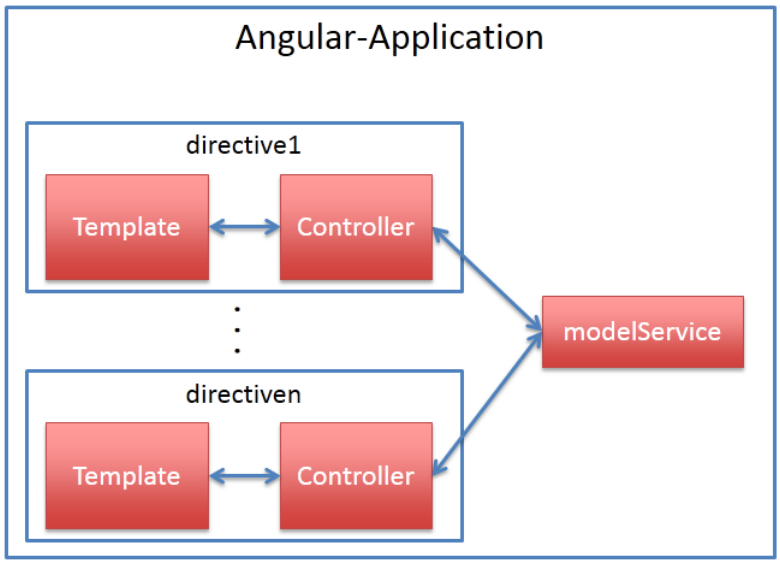


Figure 5.5: Using a well-defined interface to communicate with the application-model

5.4.2 Client-side MVC

All changes in the model have to be reflected in the View. Part of the classical observer pattern is already implemented by AngularJS. The feature Two-Way-Binding completes the MV* pattern as every event fired in the View is caught in the Controller which in turn updates the model.

Implementation. With the modifications made in the chapter above one link in the chain is missing. Updates in the Model are not reflected in the Controller. To remedy that a custom Observer-pattern is implemented: Controllers observe the global Model by the means of the "watch"-mechanism (AngularJS' implementation of the Observer-pattern).

5.4.3 Modular Structure

The new implementation of Backstage (like the old one) also makes use of the concept of modules. Every group of functionalities in the application is encapsulated in a module. The goal is to be able to exchange modules easily and have them

work without interdependencies. Another way to describe that mechanism is by understanding a module as a group of directives (which can easily have another module-Model) that communicate with a global Model. Adding this additional layer makes functionality (not only View-components) interchangeable.

Implementation. In order to achieve this property, all variables that have impact on the application in general (like View-state) have to be stored in the global Model. After finding all variables that have to be accessed, a module-interface emerges. The old Event-Hub is effectively replaced by storing module state globally. Any other module that is interested in that state only has to "watch" that value in the Model.

5.4.4 Single Point of Access to the Backend

As the Client is part of a larger MVC concept, there needs to be a mechanism of communication with the server-side. API-changes on the server-side should only have to be reflected at one point in the code providing more flexibility (see Section 3.1.1).

Implementation. This mechanism is realized by two more services: one providing a centralized interface for AJAX-calls and another for the two-way communication needed for real-time updates. Any incoming updates are exclusively reflected in the Model, outgoing calls can be made from everywhere in the application. The MVC structure propagates any received update from the server throughout the application (Figure 5.6)

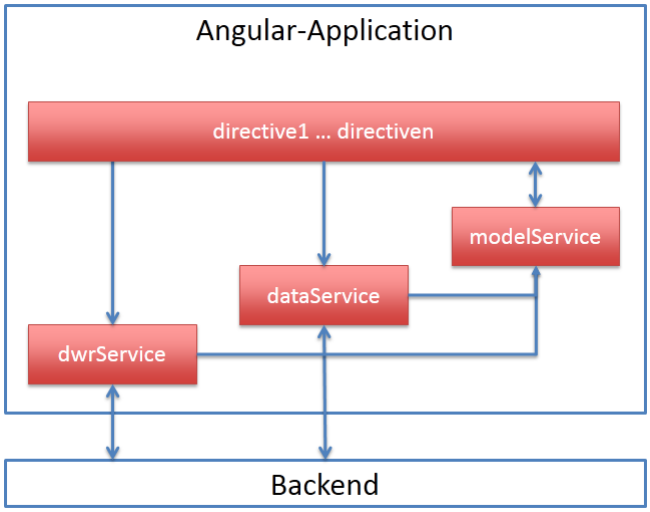


Figure 5.6: Services provide a single point of access to the backend

CHAPTER 6

Conclusion and future work

This thesis discusses Separation of Concerns as a means to obtain a maintainable and extensible software and also as a means to separate the concerns of the development team members. AngularJS, a MVC/MVP framework, is introduced to achieve a structured software design. Furthermore, criteria are explored that guide the decision towards AngularJS. To separate the concerns of the client-side developers this thesis suggests the use of tools such as the task runner Grunt for testing and deployment. Additionally, Grunt is recommended for the separation of client and server development. In a case study the RIA client of Backstage is reimplemented based on the found principles.

In the future the discussed approach could help in the development of a mobile client for Backstage (e.g. by use of standard design-libraries like Twitter Bootstrap[1]). Additionally one may wish for a more mature integration of the discussed toolchain in popular Integrated Development Environments like Eclipse[2].

[1] http://getbootstrap.com
[2] http://www.eclipse.org

CHAPTER 7

Appendix: Source Code

This thesis comprises an implementation of the Backstage client using AngularJS and the toolchain described in this thesis (Grunt task manager, jasmine tests, NodeJS as proxy and testing tool). The source code has the following layout:

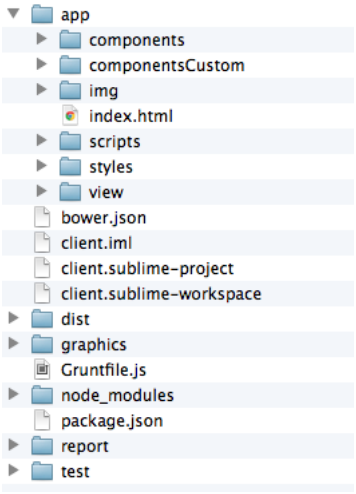


Figure 7.1: Folder structure of the source code.

The source code can be found under the branch “unverricht” in Backstage’s project repository.

Bibliography

[1] *XMLHttpRequest - Specification*. URL: http://www.w3.org/TR/XMLHttpRequest/ (visited on 04/23/2013).

[2] *Rich Internet Application*. URL: http://de.wikipedia.org/wiki/Rich_Internet_Application (visited on 08/29/2013).

[3] *Statistics: jQuery Usage*. URL: http://w3techs.com/technologies/overview/javascript_library/all (visited on 03/08/2013).

[4] *Statistics: jQuery Usage Trend*. URL: http://w3techs.com/technologies/history_overview/javascript_library/all/y (visited on 03/08/2013).

[5] Vera Gehlen-Baum et al. "Backstage – Designing a Backchannel for Large Lectures". In: *Proceedings of the European Conference on Technology Enhanced Learning, Saarbrücken, Germany (18-21 September 2012)*. 2012. URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2012-11.

[6] *Separation of Concerns*. URL: http://en.wikipedia.org/wiki/Separation_of_concerns (visited on 09/15/2013).

[7] *ISO/IEC 7498-1*. URL: http://www.ecma-international.org/activities/Communications/TG11/s020269e.pdf (visited on 10/12/2013).

[8] *Same-Origin-Policy*. URL: http://de.wikipedia.org/wiki/Same-Origin-Policy (visited on 10/13/2013).

[9] *Sencha*. URL: http://www.sencha.com/ (visited on 08/29/2013).

[10] Erich Gamma et al. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[11] *Model View Controller*. URL: http://de.wikipedia.org/wiki/Model_View_Controller (visited on 08/29/2013).

[12] *Supervising Controller*. URL: http://martinfowler.com/eaaDev/SupervisingPresenter.html (visited on 08/29/2013).

[13] *GUI Architectures*. URL: http://martinfowler.com/eaaDev/uiArchs.html (visited on 08/29/2013).

[14] *Inversion of Control Containers and the Dependency Injection pattern*. URL: http://martinfowler.com/articles/injection.html#SeparatingConfigurationFromUse (visited on 08/29/2013).

[15] *Inversion of Control*. URL: http://de.wikipedia.org/wiki/Inversion_of_Control (visited on 08/29/2013).

[16] *Notes On AngularJS Scope Lifecycle*. URL: http://onehungrymind.com/notes-on-angularjs-scope-life-cycle/ (visited on 08/27/2013).

[17] *data-binding in AngularJS*. URL: http://stackoverflow.com/questions/9682092/databinding-in-angularjs (visited on 08/27/2013).