Integration of Web Applications as cids renderer into the Navigator

cismet GmbH

IT Park Saarland
Altenkesseler Strasse 17 D2
66115 Saarbrücken

htw saar cismet

Master Thesis
im Studiengang Praktische Informatik

Integration of web application as cids renderer into the Navigator

Daniel Meiers
Matriculation No.: 3538990
daniel.meiers@cismet.de

First Examiner:	Prof. Dr. Reiner Güttler
Second Examiner:	Prof. Dr. Ralf Denzer
Advisors:	Thorsten Hell,M.SC.

February 12, 2014

Eidesstattliche Erklärung

Ich erkläre an Eides Statt, dass ich die vorliegende Master-Thesis mit dem Titel:

Integration of web application as cids renderer into the Navigator

selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie alle wörtlich oder sinngemäß übernommenen Stellen in der Arbeit gekennzeichnet habe. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in anderen Prüfungsverfahren vorgelegt. Ferner gestatte ich der Hochschule für Technik und Wirtschaft des Saarlandes, die beiliegende Bachelor-Arbeit unter Beachtung insbesondere urheber-, datenschutz- und wettbewerbsrechtlicher Vorschriften für Lehre und Forschung zu nutzen.

Saarbrücken, February 12, 2014

Abstract

The presented work describes the extension of the cids navigator, the default user interface of the cids platform, with the ability to allow an integration of web applications as cids renderer and editors. Cids renderer and editors are GUI components that provide the user with relevant information for a special object of interest. With the integration of web applications that can cover the same functionality, totally new possibilities in designing systems based on the cids platform are reached. Renderer and editor components based on web technologies offer the chance to reuse them for developing web applications that can profit from cids platform features. Hence the cids platform is totally Java based, it is not possible to supplement cids based systems with web applications. Web renderer and editor components are one necessary step to overcome this limitation.

The outlined approach for integrating web applications is highly flexible. No assumption on the type of the web application that can be integrated is made. Just the degree how seamless the integration of the web application is shaped, depends on the type of the web application. The presented solution in this respect is very scalable and can reach from the simple display of web pages up to highly integrated renderer and editor components, that are embedded in a seamless manner. Especially the latter case is proved with a custom developed web application that replaces the functionality of an already existing renderer and editor component for survey plans.

Furthermore this work provides a examination of the nowadays existing web technologies which are analysed in regard for developing web renderer components described above. The focus is put on modern JavaScript MVC frameworks, for whose an extensive and detailed analysis is given and a framework for web development projects is examined.

Contents

1 Introduction

1.1 Problem Description

The cids product suite is a platform for "building information systems with a special focus on interactive geo-spatial systems" [1] that need to operate on a combination of heterogeneous and distributed data sets. It "[..] consists of a set of services, applications, software components, management tools, development tools, and application programming interfaces (APIs)[...]" [1] that already provide "[...] a considerable number of functionalities required for complex geospatial information systems" [1] such as user management, access control, search and discovery and data visualisation. Figure 1.1 depicts the important building blocks of the cids platform.

Cids is implemented as a distributed system with a client server architecture. The server components (see blue box in Figure 1.1) are able to integrate and describe arbitrary objects from many heterogeneous data sources and offer those data in a standardized format to the clients. Furthermore, the server components are responsible for managing users, user-groups, permissions and many other properties as well as search and discovery of information.

The yellow building block in Figure 1.1 represents the client side of the cids platform, the cids Navigator and its main components. It is the default client for user interactions with the system. Among other things, the cids Navigator conjuncts information of the distributed server network and provides it to the user in a user-specific tree view the so called catalogue. For every object in the catalogue exists a so called renderer and editor. They provide a graphical user interface for the visualisation of object related information in a user friendly way and allow the user to edit these information.

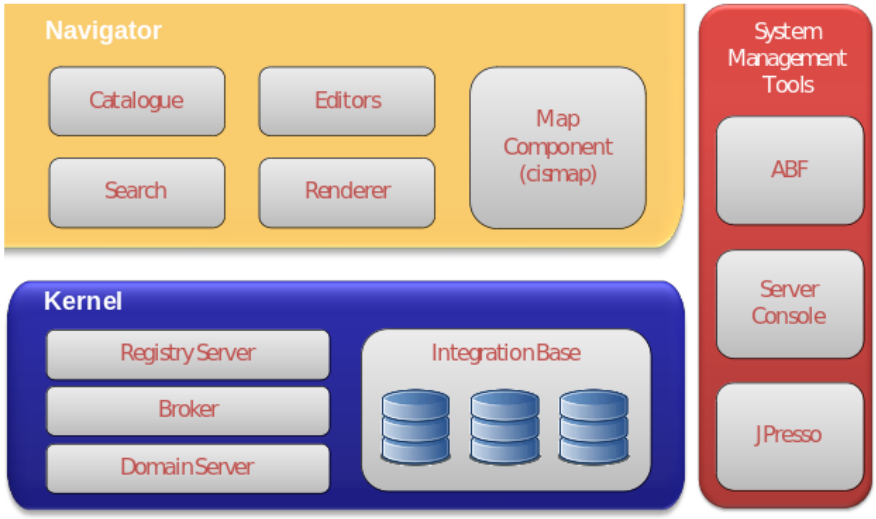


Figure 1.1: cids building blocks [1]

Just for the sake of completeness, the last building block consists of a set of management tools, that "support system managers during the installation and maintenance" [1] of cids based systems but are less important for the following considerations.

Up to now, the cids product suite, which is totally Java based, is not able to offer its services and features to other applications. Especially the lack of building web applications that uses cids server functionality is a more and more growing issue regarding the incessantly importance of the internet and its more widespread and pervasive usage nowadays. Therefore one of the latest innovations in the cids platform is a RESTful server API, that offers server side cids functionality to applications in a platform independent way. This is a immense advancement of the cids platform, hence it creates completely new possibilities in the design of cids based systems and totally new fields of application.

As already mentioned one of the most growing issues is to supplement cids systems with web applications for visualising and editing subsets of the managed data, hence web applications are much more lightweight and thus, easier accessible. The following example demonstrates this impressively. A mobile web application can be used for capturing relevant data on site. Those data could be post-processed with the already existing cids Navigator in a second step. The rehashed data (or just an interesting subset of it) can then be accessed with the same or an

additional web application. Regarding this example in more detail, it is obvious that such web applications have the same purpose as the already existing Swing based object renderer.

The similar functionality of such html applications and Swing based renderer and editors in the navigator offer the chance to use those web applications as a replacement for the Swing based editors. This has the benefit of omitting development costs for additional Swing components if an web application with the same purpose is needed anyway. Thinking one step ahead, it is also imaginable that renderer and editors are solely realized as html applications hence this offers the advantage of a much easier and more spreaded accessibility.Furthermore, the very narrow functionality of the cids RESTful server API, due to its very early stage of development, interferes initial efforts in web application development. Those efforts are very important and vital at this early phase, hence the cismet company has only slight knowledge in that field of application.

Besides these more general and strategic arguments, there is also a specific requirement in one of the EU projects, the cismet company is involved. Within the CRISMA project many html widgets are implemented that shall simplify the development of decision support systems for Crisis management. A seamless integration of those html widgets in the cids Navigator is needed.

1.2 Objectives

Up to now the cids navigator is not able to visualise html applications as object renderer and object editors. This feature is inevitable for the above mentioned reasons. Thus, the cids navigator needs to be extended with the respective functionality. One objective of the presented work is the conceptual design and technical realisation of this feature. Without going into great detail, it is obvious that this task requires a software component, that is able to parse and visualise html documents similar to existing web browsers. Furthermore, it is necessary to develop a simple web application that can act as a renderer / editor for a cids objects which simplifies testing and can act as a demo application. The development of such an first web application is the second objective of this work.

Although it is conceivable (and with the outlined feature quite possible) to use and visualise arbitrary and already existing web applications as renderer and editor components it is not clear if the integration of such "legacy" application is expedient. Hence the web application is embedded in the Swing based navigator GUI, it is important to focus on user experience and usability to allow a preferably seamless integration. We must ensure that they support a same level of user experience. Much more important is that the web application needs to interact with the navigator GUI, because some events in the Swing GUI can have effects on the we application and vice versa. For example if an object in the navigator tree is is selected, the web application needs to know which object was selected to offer the proper information.

Owing to this, it seems that the usage of custom developed web applications that regard the usage within an Swing application is necessary. An appropriate technology for building such applications is needed. Therefore the last objective is to examine the state of the art in web development with a special focus on modern JavaScript MVC frameworks and their capabilities of building rich internet applications (RIA's). In short, a RIA is a web application that behaves like an native desktop application. This examination shall reveal a framework that can be used as company wide default framework for any future web development projects.

Regarding these objectives, the thesis has the following structure: In the next chapter, the theoretical fundamentals of rich internet applications and the state of the art in developing RIA's is given. Chapter 3 outlines the conceptual design of the new feature "html applications as object renderer" and defines requirements for necessary third party web browser components and JavaScript frameworks. Based on these requirements, possible candidates are examined and compared in chapter 4. The documentation of the implementation process is stranded in chapter 5. The last chapter discusses the achieved state and outlines possibilities for future work.

1.3 Conventions

The following typographic styles are applied in this thesis.

- `Monospace` type is applied for terms belonging to code, for example
  - Class names
  - Method and function names
  - variable names
- *Italic* type is applied for
  - Emphasizing

2 Brief examination on Web Technologies

In this chapter a brief overview of the state of the art in web development is given. The different available technologies are discussed in regard to their abilities of building web application with rich interaction possibilities similar to native desktop applications and the integration in the navigator GUI in particular. This shall reveal the most appropriate technology for building web applications that can be integrated in a seamless manner.

The technologies, this comprises programming languages, standards, protocols and so on that can be used for building web applications are manifold. For example there are PHP, Python, Perl, Ruby,Java, .Net, HTML,CSS and JavaScript. To build a web application, normally a combination of multiple of these technologies are used. This makes it very difficult to select the proper combination of technologies. The situation gets even worse, if the existing web application frameworks, that are usually used when building more complex and professional web applications, are taken into consideration. A good overview over existing frameworks, that gives an impression of the variety, can be found at [2].

Despite the diversity of different technologies, the initial architecture of the world wide web is still unchanged. It allows a characterisation and classification of the different technologies. The underlying architecture of the world wide web is a client-server architecture which allows the separation of the various technologies in client and server side technologies. Picture 2.1 depicts this separation and lists a (subset) of web technologies.

The client server architecture of the web also dictates the basic communication process. The initial idea of the world wide web is that the client requests a static

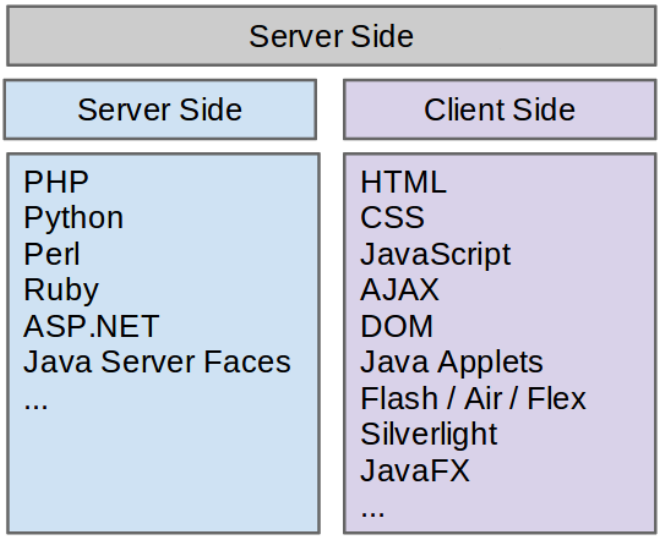


Figure 2.1: usage client side web technologies

web page from a web server. This static document contains styling information in form of HTML that can be interpreted by the client, the web browser, and is used to visualise the document. For nearly every user interaction, the client sends a new request and the process repeats. First with the introduction of Ajax by Jessie James Garret [3] it is initially possible to load data asynchronous and programatically from the server and to refresh the current page partially. Ajax is a client side technology hence it allows the client to send a request and process its result asynchronously. Usually Ajax works tightly together with JavaScript.

A pure server-side application uses a programming language like PHP, Perl or Ruby to dynamically generate a HTML document that is send to the client. The client still just has to interpret and visualise static HTML documents. The total application and business logic is handled by the server and every click on the page forces a reload of the total page. All these points are disadvantageous when trying to integrate server driven web applications into a native Swing GUI. Refreshing the whole application after every user interaction is in high contrast with the partial refresh of native desktop applications and needs to be avoided for a uniform user experience. Current web frameworks admittedly offer ways to avoid this but only by using client side scripting and Ajax. Ruby on Rails is a good example for that. Although Ruby is a server side programming language and Rails a server side web framework, it brings support for prototype.js which

is a JavaScript framework. Thus it possible to write dynamic and user friendly web applications that can be highly interactive.

Another even more important issue is, that always an additional web server is required, which is responsible for data management, application logic and session management which makes the interaction with the navigator GUI more difficult. Just to give an example: How can the navigator be informed about data changes made in the web application? Furthermore, the most server side frameworks are intended to built full stack applications which means that they are also facilitate the creation of data models, persistence, authentication and son on. This doesn't really match the situation, since those functionality is already implemented in the cids server respectively the new cids RESTful API which always provides a fix interface for the client.

As already mentioned, rich user experience and interaction possibilities can only be implemented with client side technologies. Therefore it is inevitable to concentrate on client side technologies when developing web applications that shall be integrated into the navigator.

For a long, time significant differences in the user interaction of web applications and native desktop applications exist. This is the reason why technologies like Adobe Flash or Java Applets came up, coining a totally new kind of web application, that should leave the drawbacks behind. With the increasing support of modern web standards such as HTML 5 , CSS3 and JavaScript in browsers, this limitation of native web applications fades away more and more. What all those technologies have in common, is that they allow to build rich internet applications, that offer the same level of user experience than native web applications. The following chapter therefore outlines and compares the different technologies that are available for building rich internet applications.

2.1 Rich Internet Applications

2.1.1 Definition and Characteristics

The term Rich Internet Application (RIA) was firstly introduced by Jeremy Allaire. In [4] he describes important characteristics of rich internet applications such as a powerful and extensible model for interactivity and using web and data services provided by application serves. The term RIA is not standardized and there exist various definitions often with only a slight difference.

Bozzon et al state that RIAs are "[...] a variant of Web-based systems providing sophisticated interfaces for representing complex processes and data, minimizing client-server data transfers and moving the interaction and presentation layers from the server to the client." [5]

A good discussion on the various definitions can be found at [6]. They conclude that there are two parts that distinguishes RIA from "normal" web applications, a technical part and the user experience. The technical differences are first and foremost the asynchronous data exchange with the server and the shift of application logic to the client. The second part comprises the fact, that RIA's look and behave more like native desktop applications, in fact they have a richer user interaction and can be used on and offline.

2.1.2 Technologies

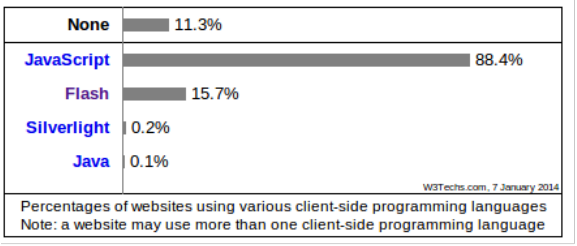
There are two groups of technologies with that rich internet applications can be build and that have evolved with the progress of client side web development technologies. Bozzon et al [7] defines 4 categories of RIA technologies,scripting-based, plugin-based (Flash, Silverlight), browser-based (XUL, XAML) and web-based desktop technologies (Java Web Start). A similar categorization can also be found in [8]. The definition of web-based desktop technologies like Java Web-Start as a RIA technology should be called into question. Such applications are normal desktop applications, in case of Java Web Start Swing applications that are not executed in the browser an require a special runtime environment. They merely use the web as deployment mechanism but can not be regarded as web

applications. Browser based RIA are more seldom which may be caused in the fact that they only run in a specific browser. Both, browser based and web-based desktop technolgies, are not suited for building web applications that can be used in the Swing based navigator GUI and as normal web application.

The more interesting types are plugin-based and sripting-based RIA's. Plugin-based RIA's, as the name suggest, need a special and often vendor specific browser plugin or runtime environment. Prominent candidates are Adobe Flash, Adobe Air, Microsoft Silverlight and JavaFX. In [9] and [10] a good comparison of the most prominent plugin based RIA technologies can be found.

There are many issues with plugin based RIA's platforms. The most important one is, that always an additional browser plugin is needed. This limits the accessibility of web application and is contradictionary with the easy and wide spread acessibility of web applications. Depending on the Platform it can happen that no plugin for every browser or operating system is available. And even if there is a corresponding plugin available, it can not be guaranteed that the plugin is installed. A good example for this are mobile devices. There is no support for Flash on any iOS device. In [11] Steve Jobs explain the reasons for this. Some of the reason he mention, can also be transferred to other plugin based RIA platforms. For example Jobs criticized that Flash "[...] has not performed well on mobile devices" [11]. This issue belongs to all plugin based RIA platforms and and is not strictly limited to mobile devices. They need a long time to load and initialize the application. In addition, there is a considerably increased security risk with one or more browser plugins installed. Flash in particular is well known for its security vulnerabilities. Last but not least, most of the above mentioned platforms are not open an doesn't belong to a web standard.

Regarding all these issues and the latest improvements in web standards such as HTML5, CSS3 and JavaScript it is not surprising that the stake of web applications that use a plugin based RIA platform has decreased rapidly in the last years. Figure 2.2 depicts the current usage of Flash and Silverlight and gives a back sight for the last three years. From special interest is the decreasing stake of Adobe Flash, hence it is the most prominent and disseminated plugin based RIA technology.



	2011 1 Jan	2012 1 Jan	2013 1 Jan	2014 1 Jan
None	11.0%	8.6%	7.4%	11.2%
JavaScript	88.4%	91.1%	92.4%	88.6%
Flash	28.5%	25.6%	21.1%	15.7%
Silverlight	0.3%	0.3%	0.2%	0.2%
Java	0.1%	0.2%	0.2%	0.1%

Figure 2.2: usage client side web technologies [12]

The last group of RIA's are solely implemented with web standards such as HTML5, CSS and JavaScript. AJAX and JavaScript are the most important technologies for building web applications with a rich user interface in fact AJAX fundamentally changes the way the client communicates with the server. The crucial advantage is that no special plugin or runtime environment is required. Scripting-based RIA's work in all standard conform web browser and on all devices. This ensures that they have the same accessibility like normal web applications. For the sake of completeness it is important to note that it is possible in the most browser to disable the execution of JavaScript. But the pervasive and wide spread usage nowadays. A investigation in 2013 of Yahoo regrading this topic [13], claims that the maximum rate measured of users with disabled JavaScript support is roughly 2 percent. Regarding those facts, it seems that this argument can be neglected.

The immense improvements in the last years regarding new standards like HTML 5, CSS3 and JavaScript remedy a lot of the till then valid drawbacks of scripting based RIA's. Just to give an example: Although they totally rely on standardized technologies, there are many differences how these standards are implemented in the different browsers, why it was often necessary to write browser depended code. The principal issue still exists today, but owing to the presence of frameworks that abstract from the underlying browser those problems are avoided in the most cases. Additionally, the rapid performance improvements of JavaScript engines in the last years, remedy the performance problem of pure script-based RIA's. Another problem of script-based applications is, that they didn't have the same level of interaction richness. But also this changed with the progresses that are achieved with JavaScript. If we regard web applications like Google Mail, Facebook and Twitter just to name a few, it is clear that it is possible to

write rich web applications solely with web standards and that behave and act like native desktop applications. Upcoming technologies like local storage and client side data bases will even allow to develop web applications that can also be used without a connection to the internet. Web Sockets will allow a bi-directional communication between the server and the client that can also be originated from the server and Web Workers will extend JavaScript with an asynchronous execution model. Summing up, scripting based RIA's seems to be the most suitable approach for building web applications that can be integrated into the navigator. They offer a fairly good level of interactivity and the simple architecture will not additionally complicate the data exchange with the navigator.

As already mentioned there are many efforts in building JavaScript frameworks and libraries that shall facilitate the development of rich JavaScript and HTML5 based web applications. In the last year, a new emerging trend in that sector are frameworks that use the MVC pattern to structure the code, keep it maintainable and increase the development productivity. This stands in contrast to the more matured frameworks like JQuery, YUI and Prototype which rather focuses on hiding browser specific implementation details and the DOM. A detailed comparison of the actual existing JavaScript MVC is given in chapter 4.2.2. The next chapter however, shall give a more conceptual overview over these new kind of frameworks.

2.2 JavaScript MVC-Frameworks

Modern JavaScript Frameworks like AngularJS, Ember and Knockout try to structure the code by using architectural patterns like MVC. In order of doing this, they achieve a separation of presentation logic, business logic and presentation state. Furthermore this seperation of concerns help developers to understand the framework much faster and get more comfortable with the it hence the used patterns provides them with a well known mental model. Unfortunately not all frameworks use the classical MVC pattern as it is defined in [14]. Some of them use slightly variants of this pattern like MVVM (Model-View-ViewModel or MVP (Model View Presenter). Knockout for example uses the MVVM pattern to enable a loose coupling of the domain model and the view (cf. [15]).

The usage of the different pattern variations often lead to discussions of the various advantages and disadvantages of the used pattern and shifted the focus away from the frameworks itself. But in fact all of the used patterns basically entail the same advantages, an Angular developer introduced the term MVW. In he argues “Having said, I’d rather see developers build kick-ass apps that are well-designed and follow separation of concerns, than see them waste time arguing about MV* nonsense. And for this reason, I hereby declare AngularJS to be MVW framework - Model-View-Whatever. Where Whatever stands for "whatever works for you".” [16]. This opinion finds more and more acceptance.

Another important feature that can be seen as a key concept of the new frameworks is automatic (two way) data binding between the model and the view. This means that it is possible to bind UI elements like text, or input fields to properties of the correlating model. Changes in the UI, for example user input, is automatically reflected to the model property, and programmatically changing the property updates the user interface.Figure 2.3 demonstrates this. This is an immense advancement in contrast to more ancient frameworks in fact the developer isn’t responsible to write and test code for that purpose by hand or need to ensure manually that changes are reflected. While there are multiple ways to achieve this, there are mainly two different approaches how the two way data binding is implemented. A good and detailed examination of those implementations can be found in [17]. One approach is to use special JavaScript objects on the model site that are able to notify observers (the view) about changes. This approach is used for example by EmberJS and Knockout. The second approach allows to bind directly to plain old JavaScript objects. Hence at the current JavaScript version it is not possible to get notified about changes of an objects value, a different approach is needed to detect changes. The technique used there is periodically checking if any of the bound JavaScript objects has changed and updating the view for every changed object. This process is often called Dirty Checking. The two approaches have some fundamental consequences that are discussed into more detail in chapter 4.2.2

Very closely related to the two way data binding feature is the usage of template engines in modern JavScript MVC frameworks. Templates support the developer in separating style and design from application and business logic. Furthermore

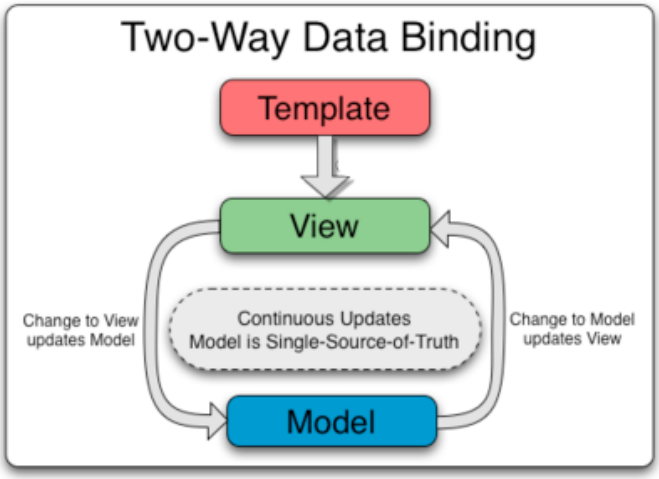


Figure 2.3: Two Way Data Binding [18]

they make it unnecessary to programmatically generate html markup. The template consists of html markup and some further place holders or variables where the final data gets inserted.

The template engines differ in the possibilities to integrate logic inside the templates. Some allow full and arbitrary JavaScript logic, other a small set of constructs offered by the template engine itself, like loops, conditionals or partials. The latter approach is the one that is mostly used, hence using arbitrary JavaScript in the templates is contrast with the initial idea of separating html markup and application logic.

Shifting the application logic to the client and only fetching data from the client creates also some new problems. Hence the total application is loaded with the first request, no entries are added to the browsers history during the usage of the web application. This destroys the browsers history and with it important functions like using the back and forward button or bookmarking certain states of the application. This problem leads us the next core component that most of the modern frameworks provide, the routing mechanism. Routing means that the different states of the application are mapped into an URL that can be used for the browsers history. The mostly used approach nowadays is, to use the "#" fragment identifier of the URL (cf. [19]). The fragment identifier usually refers to an HTML element id or a named anchor in the current page and allows to

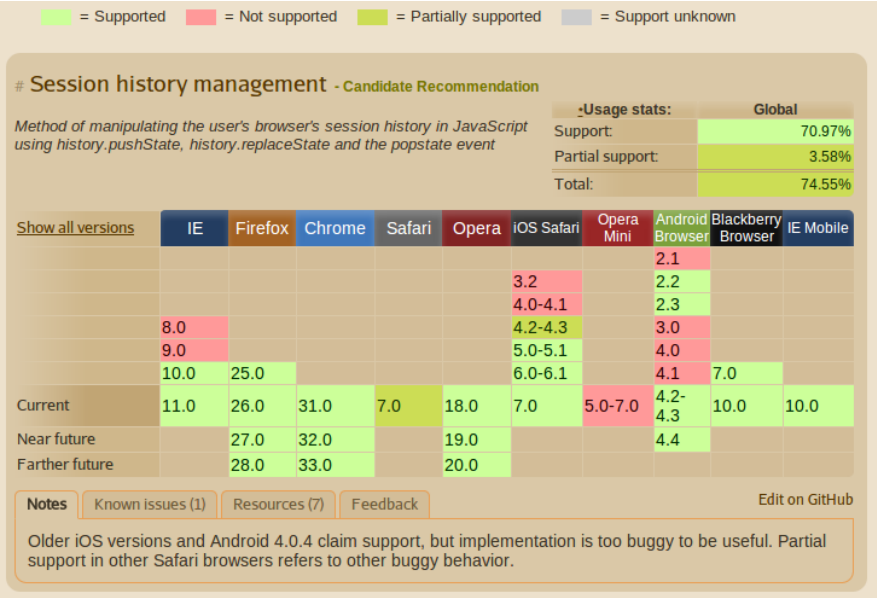


Figure 2.4: Browser Support History API [20]

automatically scroll to the position of that HTML element. Current JavaScript frameworks use the fragment identifier to track the state of the application and the router components are responsible for turning the fragment identifier part of the URL to the right application state.

Using the fragment identifier for tracking application state is highly controversial. There are a lot of blog posts and comments that discusses this approach (cf. [21],[22],[23]). To put the discussion in a nutshell the main arguments against this approach are, that it changes the target of the url, hence the fragment identifier is not send to the server, and only clients with JavaScript enabled can interpret these kind of URLs. Using hash-bang URLs was the only possibility for tracking state in client side JavaScript applications. Fortunately the upcoming HTML 5 standard contains a new part, the History API [24], that allows the browser to modify the current URL and the browser history programmatically. The History API offers a new way for tracking application state without using hash-bang URLs and the related drawbacks. Hence most browsers already support the History API (see also figure 2.4) it is just a matter of time till this feature replace hash-bang URLs in the frameworks. Angular for example already offers the possibility to already use the History API (cf. [25]).

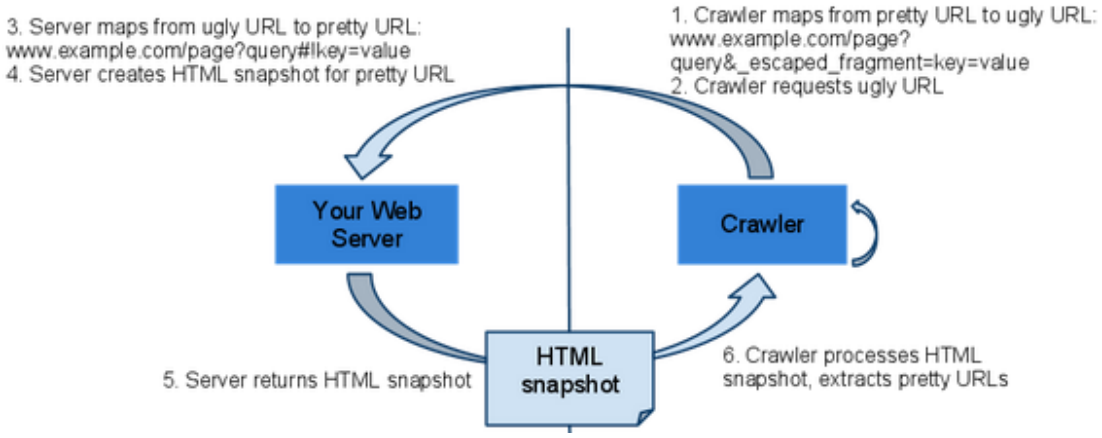


Figure 2.5: Google crawling AJAX [26]

Another problem that comes up using JavaScript driven web applications is that they can't be analysed from search engines as easily as server centric web applications because the initial page request doesn't return a HTML document that contains the important content directly. But also for this drawback certain technologies exists, that remedy this disadvantage. The first on was announced from Google in 2009. In [26] they announce a method to crawl hash-bang URLs. The main idea is that the crawler maps the hash-bang URL into an normal URL that contains the fragment identifier. Figure 2.5 depicts the idea. Important to note is that the web server needs to deliver a custom and search engine optimized HTML document for the URL. This is what all approaches have in common. An other upcoming approach, that gets more and more popular, is to use a headless web browser like PhantomJS on server side that is able to parse the web application like a normal browser and can output the normal HTML markup to the search bot.

3 Conception

After the analysis of the state of the art in web front end development in general and JavaScript MVC frameworks in particular, this chapter outlines a conceptual approach how new web renderer components can be realized and can be integrated in the cids system, or more precisely, in the cids navigator.

As already mentioned it is obviously necessary that a swing browser component is needed for visualising arbitrary web applications in the navigator swing GUI. Hence an own implementation would be totally out of the scope if this work, it is necessary to search and examine respective third party components. This chapter lists and explains requirements for candidate components and API's. Based on these these requirements the examination of the various candidates is done in chapter 4.1.

Hence another objective of this work is to evaluate a modern JavaScript APIs for a company wide use in general and for the development of an demo web application in special, this chapter also outlines requirements that can be used for the evaluation in chapter 4.2.2.

3.1 Overview

The main goal of this work is to extend the existing navigator with the ability to integrate web applications that can be used as cids renderer and editor components. To fully understand the consequences and implications of that undertaking and to deduce requirements it is necessary to have a more specific look what cids renderer and editors are and what tasks they assume in the cids navigator.

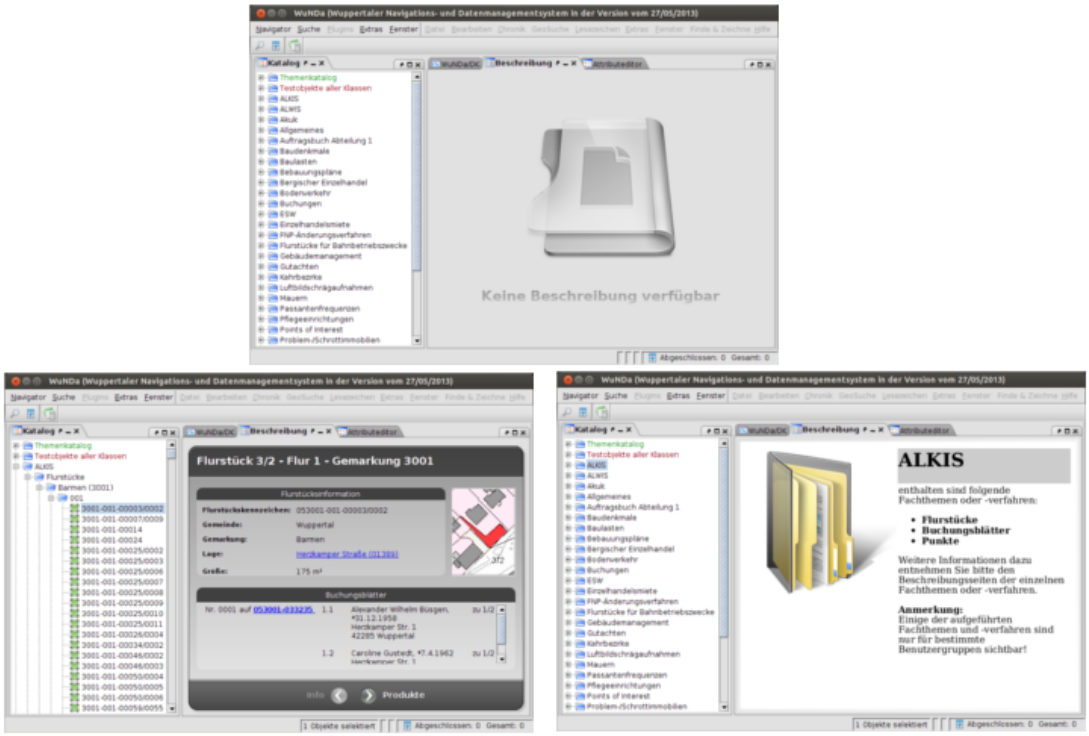


Figure 3.1: Navigator catalogue and different description panes

One basic component of the cids navigator is the catalogue tree. The catalogue tree offers a uniform and user specific tree view of the integrated information in cids. The tree is used "for navigation of the distributed catalogue and provides basic information about objects" [1]. For every node in the catalogue tree, it is possible to offer more specific information to the node. Those information are visualised in another part of the navigator called the description pane. Figure 3.1 depicts an example catalogue with various contents of the description panes can have.

Cids renderer are responsible for offering the user a graphical user interface that renders all the relating information of the specific object node that is selected in the cids catalogue. The kind of the renderer that is used, is determined on a per-class basis. The renderer only allows the visualisation of information, for editing those information cids editors are used. They follow the same principles as cids renderer but are placed in another panel within the cids navigator.

Regarding this basic setup it is clear that if we want to use web application as

cids renderer, they need to follow the same per-class approach like the existing one. This means that the web application itself must be parametrized with the information which concrete object is currently selected in the navigator and then needs to visualise the information for that object. For web editor components, it is also necessary to provide the edited information in a way that the navigator can access them.

This interaction with the web application is quite essential hence only in that way a seamless integration of the web application into the navigator GUI can be reached. Although a possibly seamless integration of the web application is worthwhile, this always requires specific adoptions to the web application to make the interaction possible. But it is also a quite imaginable and reasonable scenario to integrate legacy applications where such an seamless integration can not be afforded. For example it is imaginable to use a legacy web based information system as per-class renderer and editor. In that scenario no data exchange between the web application and the navigator is necessary in fact the web application itself takes care about all objects. Therefore, the technical realisation should not limit the kind of web application that can be integrated, if possible.

Nonetheless, the implementation should focus a possibly seamless integration. The following consideration try to outline possible ways that achieve this. For the data exchange between the navigator and the web application different approaches are available. We can distinct the approaches according to initiator of the data exchange.

It is conceivable that the loaded web application tries to fetch the currently selected object from the navigator. The problem with that approach is the web application does not know when the user has selected catalogue node, in fact this happens out of its context. Furthermore, the current implementation provides that the cidsBean is injected into the renderer and editors. Following this approach would stand in contrast with the current implementation and causes more efforts to implement.

The opposite is, that the navigator ships the information to web application. This would avoid the above mentioned drawbacks and would better reflect the current implementation moreover. As mentioned above, if a object node gets selected, the

navigator loads the corresponding renderer component and injects the currently selected object into it. Transferring the same approach to web applications would therefore cause less implementation changes. The question is how can the Java based navigator inject data into the web application?

One solution is to parametrize the URL that is used to retrieve the web application with the relevant information. Using this approach requires that the web application is hosted by a web server, which is not necessary in every case. If the web application doesn't need any server side processing it is also possible that the web application can be packed in one of the jar files necessary to run the navigator application. Furthermore additional logic on server side is required to process the URL encoded data. Much more critical is, that the relevant information of the selected navigator object needs to be encoded into URL. This encoding has some limitations. First, it is not possible to encode binary data for example. This can not be excluded hence in WuNDa for example already renderer exists that also need to visualise binary data like pdf documents or images. Another problem is that the information that needs to be encoded can be very large and can have a complex structure. Although the RFC [27] does not limit the length of an URL, there are practical limitations depending on the browser and web server used (cf. [28]). Additionally, using this approach would also complicates the reuse in other contexts.

The second option is, that the navigator injects the selected option on client side, when the application gets loaded. This requires, that the navigator must be able to execute JavaScript code within the context of the web renderer component and that the web application can make method calls to Java. Fortunately both ways are possible with the Java Scripting API [29]. The problem is, that the web application shall be rendered by an third party component. Using this approach requires that the Java browser component offers an interface for communicating with the JavaScript environment of the loaded web application and vice versa.

Web renderer components that shall be integrated in the navigator can be wrapped with some JavaScript logic that retrieves the data that should be displayed. This simplifies the reuse in any other web application hence only this part of the application needs to be re-factored.

Another basic advantage of using web applications as cids renderer is that they can be reused when building web based information systems that make use of cids features and the new cids REST API. Therefore, it shall be possible to reuse the developed web applications with at least efforts as possible.

Last but not least, the effort for the configuration which objects should use an web application as cids renderer shall be as small as possible.

The following list summarizes the requirements for the technical realisation and the further considerations.

Requirements:

1. no limitation that restricts the kind of web application that is integrated ?
2. but integration must also be as seamless as possible
3. Data exchange between navigator and web application
4. It should be possible to easily use the web application also without the navigato
5. Only one code base for editor and renderer
6. There should be an easy way to configure the usage of the web application

3.2 Integration in cids

After giving an overview what the expected features of the new developed web renderer component are, and discussing some general and architectural approaches, this chapter outlines the integration in the cids navigator in a more detailed and technical manner.

The first issue which needs to be clarified for integrating web applications as cids renderer into the navigator is, how the configuration can be modeled. As defined in the requirements above this configuration should be as easy as possible. Furthermore the configuration needs to be done on a per class level. Especially the per class constraint offers the chance to use cids class attributes to easily solve this issue. Class attributes are part of the cids meta information system

and provide a per class key-value store, which can be used to provide class wide settings. Using class attributes allows an easy implementation of the new feature. What is just necessary, is to define a new class attribute that indicates the usage of an web application as renderer and editor. The generation of class attributes is already supported by the ABF managing tool. Hence, an uncomplicated and tool supported approach for the configuration is ensured.

Now that the indication when to use a web application is clear, the next question to answer is how to point to the location of the web application and where the applications are stored. Therefore we have a closer look on the existing mechanism that cids already provides for determining what gets displayed in the description pane.

The information that gets displayed in the description pane depends on the type of the selected catalogue node and the options that are configured for it. There are two types of catalogue nodes, object and organisational nodes. Organisational nodes allow a categorisation of the existing objects nodes after arbitrary aspects. For them it is possible to configure a static html page that can offer more information about the selected node. An object node represents one object that is part of the cids information system and is from particular interest for the user. If an object node is selected, the navigator loads the corresponding cids renderer into the description pane based on a naming convention. This naming convention relates the class name of the selected object to an Java class within a special package. The following example demonstrates this. If an object of the class “VermessungRiss” is selected the navigator loads the Java class

VermessungRissRenderer.java

from the package:

de.cismet.cids.custom.objectrenderer.wunda_blau

A naive approach could be to use and extend the naming convention in a way that it also works for web renderer. The only adjustment to the convention needed is to introduce a new package in which the web applications are stored. Transferring this to above example the path to the web application would be the following:

VermessungRissRenderer/index.html

Another important change to mention is that the Java class is replaced by a folder that contains the necessary files and the entry point (index.html) of the web application.

The problem using this approach is that the web application needs to be integrated in the sources. Owing to this, applications that needs web server functionality (e.g. a server side scripting language) can not be used. To overcome this issue it is necessary that the web applications are hosted by a dedicated web server or to use a locally started Java based web server like Jetty [30]. Starting a local web server on client side can cause other problems like missing permissions or ports that are already in used. But besides those specific issues, there are also two general problems with that solution. The first one is, that it causes additional configuration overhead hence the web server where the applications are hosted must be configurable. Another critical issue is, that it is still not possible to integrate (legacy) web applications that are hosted on any other web server. Therefore a more proper and more flexible solution is needed.

A much better solution is to use the already new introduced class attribute. Hence the class attributes act as a key-value store, the value field of the new class attribute can be used to point to the location of the web application. This approach avoids assumptions on the storage and offers a high flexibility hence it allows to integrate the web application in a certain package within the sources as well as the integration of legacy applications, which is one of the requirements defined in the chapter before.

3.3 Requirements

3.3.1 Requirements for Java Browser API

To integrate modern web applications in the cids navigator as object renderer and object editors candidate components or API's should provide as many of the following functionality as possible:

The most important requirement regarding the Java browser component is, that it must be able to execute JavaScript. This is necessary since it shall dis-

play JavaScript based Rich Internet Applications implemented with a modern JavaScript MVC framework as discussed in chapter 2. Furthermore, the candidate component needs to offer an interface for executing JavaScript code in the context of the loaded web application and vice versa. As discussed above this is the preferred way for the necessary data exchange between the cids navigator and the loaded web application.

Another important requirement is the support of the latest web standards such as HTML5 and CSS 3. Hence both technologies stand for a group of different standards, every single one with a different status and not even finished, it is impossible to make any quantitative statement about the support of both standards. Nonetheless there are different test tools available that can at least be used as an indicator for the support of new HTML and CSS features.

The last mandatory requirement regards the licensing of the candidate components. The cids navigator is developed under the GPL license which has strong copyleft conditions. Therefore, the candidate components licensing must be compatible with the GPL license.

Besides these mandatory requirements there are also some less obligatory issues candidate API should fulfil. For example an important issue is how good the API is documented. It is an big advantage if there are developer guides, tutorials or code examples that explain the usage and ensure an easy start and usage of the candidate API. Closely related to this, is the community and the support that is provided if any problems during the usage occur. Also relevant is, if there is a bug tracker system or a forum where individual problems can be posted and how responsive the developers or the community react on such issues.

The following table summarizes the various requirements and depicts what requirements are inevitable for a possible use and what requirements are optional.

Requirement	Comment	Mandatory
CSS 3 Support	as far as possible	yes
HTML5 Support	as far as possible	yes
JavaScript Support		yes
Bi directional communication		yes
GPL compatible license		yes
fast on rendering		no
documentation		no
support		no
community		no

Table 3.1: Requirements for Java Browser Components and API's

3.3.2 Requirements for JavaScript MVC Frameworks

For the evaluation of JavaScript MVC frameworks it is also necessary to define requirements, candidate frameworks should fulfil as good as possible.

The different requirements can be classified into four groups which concerns different aspects. The four classes concerns the ease and speed of development,flexibility, stability and performance and legal requirements.

The first category, ease and speed up of development, concerns to the general idea why a framework is used in general, and therefore is the most important one. The framework should reduce the amount of boilerplate code to a minimum. Especially important in that relation is that the framework supports two-way-data binding between the model and the view. This is a quite essential feature when developing cids render and editors, hence the data visualisation and manipulation is the most fundamental task of them. Furthermore, it is a benefit, if the framework provides tools for testing and debugging the developed application. Another crucial factor that influences the ease of development is the quality and the extend of the documentation as well as the size and activity of the community or the support that is offered by the framework developers. All these factors can simplify or complicate the development with the framework, especially if the framework is not very matured and shall be initially introduced.

Regarding the next category, the flexibility of the framework, the most important requirement to mention is, that the framework should not limit the set of GUI elements that can be used in any way. The Google Web Toolkit [31] is a good example for this. For example it is not possible to easily use an arbitrary JavaScript based date picker component or slide show. Previous experiences in web development projects where GWT was involved has proven that this is a major drawback and should therefore be avoided. The second requirement concerns a more general issue. Although the candidate framework should support the user in many tasks as possible and hence eases the development, but it should also allow alternatives way to achieve the same goal. A good example for this is the routing mechanism of the framework. Although it is useful if the framework already provides a way to support routing, it should not prevent the user from using a third party library for this purpose which can be more suited.

Another group of requirements concern the stability and performance of the candidate frameworks. A very common problem when developing JavaScript applications is the creation of memory leaks, hence there is no global garbage collector. The frameworks itself must ensure that the creation of memory leaks is prevented when building applications based on them. Furthermore also the performance of the framework is a key factor. Owing to the short time since MVC frameworks came up and the speed of development in this sector, the maturity of the framework as well as the future support and continuous development of the framework are also important.

MVC frameworks often provide solutions for features that are not possible today, but whose are already in the standardisation process. For example they provide mechanism to get notified about changes in JavaScript objects. A corresponding feature that will be integrated in the next version of the ECMAScript standard is the Object.observer() feature. Another example is the upcoming web components standard, which simplifies the development of reusable components, also a feature many MVC frameworks offer already today. To guarantee a long life-cycle of the developed applications it is important that the candidate frameworks makes transparent, how they plan to support such upcoming standards.

Finally the last requirement concerns the licence of the framework again and similar to the Java web browser API, the framework licence need to be compatible

with the GPL. Table 3.3.2 gives an overview over the requirements.

Requirement	Comment	Mandatory
Speed and Ease of Development		
Two-way data binding		yes
Reduce boilerplate code		yes
Debugging & test	are there tools available for debugging and does the framework provide a test concept	no
Documentation	a good quality and extend of the documentation is needed	no
Community	is a bug tracking system and forums available, size and responsiveness of the community	no
Flexibility		
GUI elements	The framework should *not* limit the GUI elements to a strict set	yes
Opinionated	the framework should ease the default case but *not* limit special adoptions	yes
Stability & Performance		
Memory leak preventions		no
Performance		no
Future standard support	are adoptions to future web standards planned and how transparent is this for the user	no
Legal Requirements		
License	license must be compatible with GPL	yes

Table 3.2: Requirements for JavaScript MVC Frameworks

4 Technology Analysis

4.1 Choosing Java Browser Components

There are already existing Java APIs that allows the visualisation of HTML applications in Java Swing components. They all offer the basic functionialty to parse HTML documents and visualise them in Swing components.

The following APIs are examined with regard to their ability to visualize modern web applications, which means applications that highly make use of JavaScript, CSS3 and HTML 5 features. In particular the above introduced requirements for browser APIs are taken into consideration.

In the current implementation of the cids navigator it is already possible to use simple HTML sites as a description page for catalogue nodes. It is possible to choose between two different HTML rendering API's, Flying Saucer and Calpa HTML. They are examined first in regard for a possible reuse.

CalpaHTML is the first component that can be used to visualise html documents as a description page for nodes in the cids catalogue. Despairingly, it seems that the project was abandoned because no further information (API Documentation, feature list etc) can be found. Past experiences with CalpaHTML have shown the very limited feature set. Thus, Flying Saucer as an optional HTML rendering component can be used.

Flying Saucer is an XHTML renderer that allows the rendering of XML and XHTML documents to images, pdf documents and Swing components such as JPanels or ScrollPanes as well. The main idea of Flying Saucer is to parse any kind of XML documents and to apply CSS styles to them. According to the website, Flying Saucer supports nearly all CSS 2.1 with a few exceptions and

additionally "includes a small handful of CSS 3 features" [32]. Unfortunately it is not possible to determine which CSS3 features are supported and which not in more detail. Documents parsed with Flying Saucer need to be valid and standard compliant documents. It is not possible to parse faulty documents. The developers recommend the usage of an HTML cleaner like TagSoup or JTidy, which are able to make invalid HTML documents valid. The documentation of Flying Saucer is good. There are several How-To articles as well as an developer guide that makes it easy to use Flying Saucer. Furthermore there is a mailing list where it is possible to get support from the developers and the community. Unfortunately, there is a rapid decrease in the activity of the mailing group since 2010. The most disqualifying factor is that Flying Saucer lacks in supporting JavaScript or any kind of user editable content, since the parsed documents are read only.

The **Lobo Project** offers a complete stand alone Java based web browser as an alternative to other mdoern browsers. It consists of the Cobra toolkit which is responsible for HTML parsing and rendering on the one side and a browser application on the other side. The devlopers emphasize that the Lobo browser is an "[...] browser application that has been documented so it can be used as an API." [33] and that sits on top of the Cobra toolkit. It enriches its users with additional functionality such as navigation, request handling, authentication, caching and so on. The Cobra HTML renderer and parser fully supports HTML 4, Javascript and CSS 2. HTML5 features and CSS3 are not supported. Both projects are available under a LGPLv2 license. The API is well documented, but there are only a few tutorials and how-to documents which makes it a bit harder to start. This is especially true for the Cobra toolkit. There is a help forum available but sparsely used and maintained, hence there are a lot of unanswered questions. This and the fact that the last release was in 2009 indicates that the project is shelved.

Another full Swing and embedded browser component is **WebRenderer**. It is built upon the Mozilla technology and therfore supports an impressive list of standards and features such as many HTML5 features, CSS1 and 2, JavaScript, Java Applets, SVG and MathML just to name a few. Another outstanding point in contrast to the earlier discussed frameworks is the fine granular control over the

loaded web page. WebRenderer gives full access to the DOM of the loaded site and offers a full event model which allows it to react on mouse events, browser events, network events and DOM events. The API is well documented and there are a lot of code Examples as well as an developer guide. The drawbacks of WebRenderer are a missing CSS3 support and, the more serious issue, the license. WebRenderer is commercial product and can therefore not be used.

In **JavaFX** there is a built in browser component called WebView, that is based on WebKit an open source browser engine, which is also used in modern browsers like Safari. Similar to WebRenderer there is also a high support of modern web standards. To give an example, WebView supports the following HTML5 features: Canvas, MediaPlayback, form controls, editable content and so on. Furthermore it supports JavaScript and several CSS3 features. It also offers a bi-directional communication between JavaScript code of the loaded site and JavaFX. Hence WebView is a native JavaFX component, the documentation as well as the support are very good and it is no problem to find information or get help on a special problem. The JavaFX community is large and growing, and there are several forums, among others the Oracle JavaFX forum. Since Java 7, JavaFX is part of the standard Java runtime environment and hence available platform indepent. In contrast to the Lobo project or CalpaHtml, it is unlikely that the support of the WebView component gets discontinued.

All this make the JavaFX WebView to the best suiting candidate, it supports an extended set of HTML5 features, JavaScript and CSS3, is available as open source and is well documented with a large and active community. But the usage of JavaFX emerges the problem to integrate JavaFX components in the cids Navigator respectively swing components. Fortunately JavaFX also provides a component for this task. The JFXPanel is a Swing component that acts as a JavaFX runtime container and makes it possible to run arbitrary JavaFX application scenes in it.

The following evaluation matrix summarises the pros and cons of the above introduced browser components and APIs. Disqualifying factors are marked with a coloured background

	Calpa HTML	Flying Saucer	Lobo Project	WebRende	JavaFX WebView
Java Script Support	no	no	yes	yes	yes
Access to loaded Web page	no	no	no	yes	yes
HTML5 Support	no	not clear	no	yes but uncertain extent	yes
CSS3 Support	no	small handful	no	no	yes
License	LGPL	LGPL	GPL	commercial	proprietary / GPL
Documentation	not found	good, How-To articles, Developer Guide, Examples	just a few tutorials		good, many and detailed examples, and how-To articles
Support, Community	project aban-doned	rapid decrease in activity	sparsely used forum	commercial support	goods, many forums, large community

Table 4.1: Decision Matrix Java Browser Components

4.1.1 Testing and Comparing the JavaFX WebView

Despite an exhaustive literature research it was not possible to find a full feature list of the WebView concerning modern web standards like CSS3, JavaScript or HTML5. To ensure that the WebView is able to render modern web applications, especially JavaScript based applications, correctly, it is necessary to test the compliance to web standards and the behaviour when trying to visualise JavaScript applications. This is done with several test tools that are available on the web. The following section carries out what test tools are used, gives an short description about each test tool and outlines the results of the WebView as wells as the results of the Google Chrome browser to allow a first comparison.

The first test that is executed is the ACID 3 test originated in 2008. The test is defined by the Web Standards Project and can be found at [34]. According to [34] the test "[...] is primarily testing specifications for "Web 2.0" dynamic Web applications". A full list of tested standards can be found at [34] but among others the CSS3 Color, CSS3 UI, SVG and a lot of DOM standards are tested. Important to note is, that the resulting image is not the only criteria to pass the test. Also the animation has to be smooth. As figure 4.1 depicts, the test is not fully passed. Unfortunately the test give no information what test has failed or what standard the browser is not compliant to. Besides this obvious failure, the animation smoothness compared to other browsers, especially compared to Google Chrome, is considerably worse. Another important thing to mention is, that it is recommended to execute the acid 3 test with default settings hence e.g changing the zoomlevel "[...] may alter the rendition of the test page without this constituting a failure in compliance." [34]. In fact the tests are executed with the Oracle demo application, where it is not possible to a maximum size for the applciation, it is unclear if the layouting mechanism used in JavaFX influences the test results.

To test the abilities of the WebView regarding HTML5 support, the HTML5 test [35] is executed. The test is seperated in different features that are part of the upcoming HTML5 standards as well as drafts and specifications. It tests how many of these features are already supported and calculates a score. Important to note is that the feature set that is tested is not complete and that the test

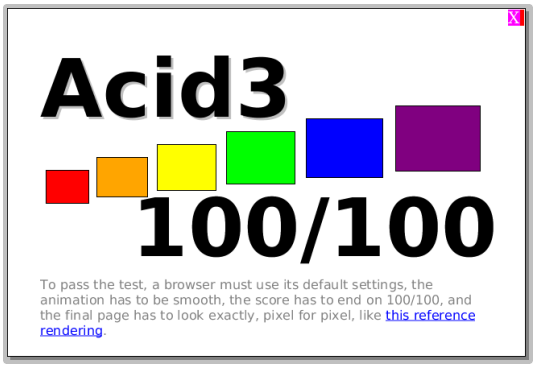


Figure 4.1: Result Image of the Acid Test - WebView browser

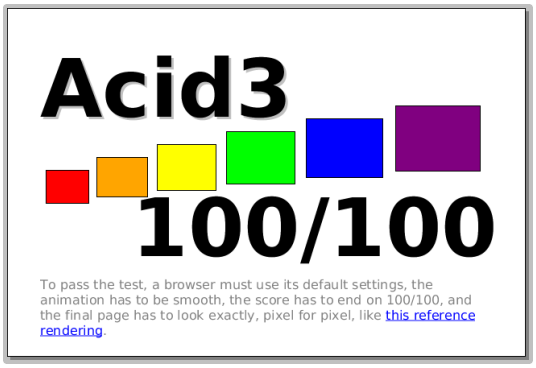


Figure 4.2: Reference Image of the Acid Test

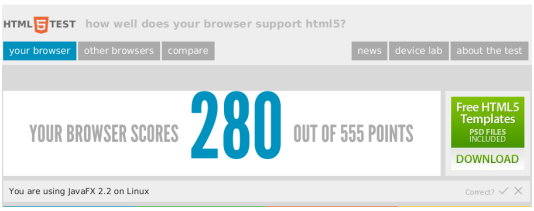


Figure 4.3: Result Image of the Acid Test - WebView browser

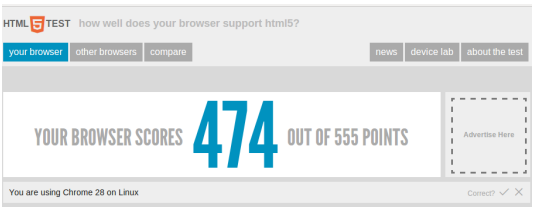


Figure 4.4: Reference Image of the Acid Test

does not check the correctness of the implementation. As already mentioned, a lot of the HTML5 standards are still under development and could change before receiving a final status. Thus, the rather poor result (see 4.3) can be relativated a bit.

As already mentioned in the requirements, it is vital that the browser supports as much CSS 3 features as possible. Hence it is not possible to find information regarding the CSS3 compatibility of the WebView, the CSS3 [36]is used to get a better insight on this issue. Similar to the HTML5 test, the test is divided into the different modules defined by the CSS3 standard. Again, the test checks only the support of the various features rather the correct implementation. Figure 4.5 depicts the result of test. In this case, the result of the WebView is comparable with the result of Google Chrome (see figure 4.6). The reason for the general poor results might be that the CSS standard is still under development.

The test suite contains thousands of individual tests, each of which tests some specific requirements of the ECMAScript specification.

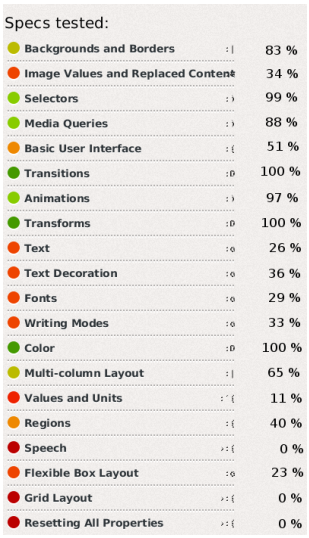


Figure 4.5: Result Image of the Acid Test - WebView browser

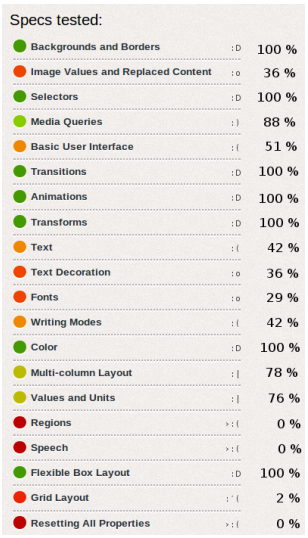


Figure 4.6: Reference Image of the Acid Test

The last thing to test are the JavaScript abilities. The ECMA International organisation, which is responsible for the standardisation of the ECMAScript language (the official name of JavaScript) provides the *Test262* test suite that can be used "[..] to check how closely a JavaScript implementation follows the ECMAScript 5th Edition Specification" [37]. It provides thousands of tests "[...]each of which tests some specific requirements of the ECMAScript Language Specification." [38]. Although the suite consist of a huge amount of test cases, the authors emphasise that "test262 is not yet complete. It is still undergoing active development." [38]. In contrast to the above mentioned tests, the ECMA test262 suite lists every single failed test and offers additional information to this test case. Thus, it provides a good documentation of the bugs that are shipped with the tested JavaScript environment. Regarding the test result of the WebView, there are 236 test cases that fails. Although this seems to be a large number, the percentage stake of failed tests is about 2%, which is a good result.

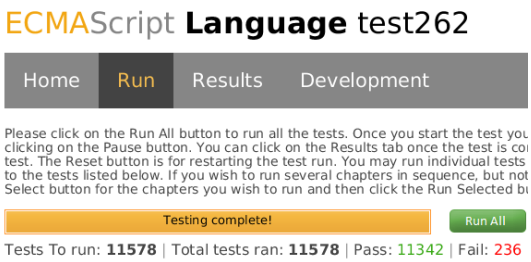


Figure 4.7: Result Image of the Acid Test - WebView browser

Figure 4.8: Reference Image of the Acid Test

4.2 Comparison of modern Java Script Frameworks

4.2.1 Overview

There is a huge and steadily increasing amount of new JS MVC frameworks. A good overview over existing projects is provided by the Todo MVC App "tech-ana:todo-mvc". The Todo MVC app was created with the intention to help developers selecting a JavaScript framework, by providing implementations of the very simple and ever same ToDo App, which allows to define a set of tasks that can be marked as done or undone.

The TodoMVC app gives also a good evidence of the rapid increase of existing JavaScript MVC frameworks. Figure 4.9 shows a screenshot of the ToDoMVC taken at two different dates with a period of a year.

The large amount of different frameworks makes it impossible to examine them all in detail. Therefore a two step approach to filter the best suiting candidate is applied. In a first step a very rough comparison of the frameworks is made to filter out the most inappropriate frameworks. The remaining frameworks are then examined in more detail in chapter 4.2.2. Furthermore a simple application is implemented with each framework that is selected for the detailed comparison to get a better understanding of th concepts used there. The implemented application is a Simple Greeting Application where a user can edit its first and last name and sees a greeting. It is obvious that the complexity of this demo application is not sufficient to demonstrate the powerfulness of the frameworks

Figure 4.9: Screenshot TodoMVC app left:July 2012, right: December 2013

or to compare them. But it allows first impressions of the concepts that are used, and how effectively writing boilerplate code is avoided.

The following list shows relevant JS MVC frameworks taken from the ToDoMVC app. The list only contains frameworks which are available in a stable release (at least 1.0.0) and available under a GPL compatible license. The TodoMVC App also lists combinations of different frameworks which are also excluded in the following list.

- Dojo
- YUI
- CanJS
- soma.js
- Maria
- Backbone JS
- AngularJS
- Ember JS
- Knockout JS

Dojo as well as YUI (Yahoo! User Interface) are not MVC frameworks in the proper meaning of an MVC Framework. Similar to JQuery , they are rather DOM/Ajax wrapping libraries which is a more suitable description for those frameworks. Both has several additional features like UI-elements, effects and animations. Both doesn't provide components that implement MVC pattern and don't provide the user with built in features like data binding, templating or routing and are therefore not taken into further considerations.

CanJS, Soma.js and Maria are very new MVC Frameworks all providing a different set of features and a different emphasis. According to [39] CanJS combines the best features of lightweight frameworks like BackboneJS, which are easy to learn and have a small size, as well as of heavy frameworks like EmberJS which offer a lot of good features like live binding,computed properties and memory safety. CanJS can use EJS or Mustache as templating engines which are both string based templating engines. In one sentence CanJS can be characterized through its high performance, its memory leak prevention and the fact that it can be used with a lot of different DOM libraries.

Although soma.js can be used as a MVC framework, the basic idea behind soma.js is a more general and architectural approach. "soma.js provides tools to create a loosely-coupled architecture broken down into smaller pieces." [40] In order to do so, it implements a large set of different design patterns, like dependency

injection, observer pattern or mediator pattern and the MVC pattern. Soma.js implements its own template engine based on soma.js. The more architectural bias makes it less comfortable for the developer to work with soma.js since a lot of work needs to be done manually that other frameworks do automatically.

The authors of Maria emphasize that the framework implements the "real" MVC pattern. Maria is a very lightweight framework which means that there is no built in support for templates and data binding. Similarly to soma.js, working with Maria means to write a lot of boilerplate code.

Generally speaking, the biggest problem with the above mentioned very young frameworks is, that these projects are not so well documented and there is only a small community which makes it very difficult to find information or help when problems occur. This disqualifies these frameworks as possible candidates.

One of the more matured and proven frameworks is Backbone JS. A lot of really large and impressive projects are built with backbone.js such as LinkedIn, AirBnB, Trello or FourSquare. Hence its maturity, it is well documented and there is a large and active community which makes it easy to find help. Backbone JS consists of different components concerning all necessary aspects of building single page web application such as models, views and routers. Thus application built with backbone normally have a clear defined structure [41]. Backbone has built in support for persisting model data to REST apis. To make this built in component to work properly it is necessary to have a correlation of your model names and the REST api paths. The backend connection can be implemented manually if the standard one is not suited. There are already some existing Modules that do this like Backbone.localstorage which stores all data in the Browser itself.

An adverse is, that Backbone JS is not able to reflect changed model data to the view automatically (and vice versa). Backbone's approach to achieve this is much more lightweight. Each model can fire a set of events. A backbone view can listen to these events. Each view has a render function, which generates the HTML for displaying the view. This function needs to be implemented manually for each view. This approach allows it to use any templating engine. Since backbone needs the dependency to underscore, you can use the templating function from

this library. Another drawback is that, backbone does not manage the display of different views. Admittedly it provides a router object, but again, this objects just defines what function should be executed when a route is loaded. The developer has to initialize the new view and to destroy all views no longer needed. There is always the danger of memory leaks which causes several performance and security issues. At least, using backbone means a lot of initialisation overhead, since it is first necessary to define the domain model and set up the infrastructure. Considering all this disqualifies Backbone JS as possible candidate.

The remaining 3 Frameworks does not have any of the above mentioned drawbacks and are used for a much more detailed examiniation described in the following chapter.

4.2.2 Comparison of AngularJS, EmberJS and KnockoutJS

Knockout JS

Knockout JS is a more leightweight library, which emphasis is to implent the MVVM (Model-View-ViewModel) pattern for HTML. This pattern enriches HTML with two way data binding and a seperation of concerns, which means that the application logic is seperated from the business logic and the model.

The two way data binding is implemented by using special Java Script Objects, so called Knockout observables and a special HTML attribute both provided by Knockout. Thus model changes are automatically reflected to the view. Listing 4.1 demonstrates how to bind a property of a ViewModel to an HTML textfield element.

Listing 4.1: the ViewModel

```
1 function ViewModel(){
2   this.firstName = ko.observable('Daniel');
3   this.lastName = ko.observable('Meiers');
4   this.fullName = ko.computed(function({
5         return this.firstName()+" "+this.lastName();
6       },this);
7 }
```

```
8 ko.applyBindings(new ViewModel());
```

Listing 4.2: the html view

```
1  <div>
2    <label>First Name</label>
3    <input type="text" data-bind="value: firstName">
4  </div>
5  <div>
6    <label>Last Name</label>
7    <input type="text" data-bind="value: lastName">
8  </div>
9  <p data-bind="text: 'Hello '+fullname"></p>
```

The data-bind attribute of the input element (see line 3 in listing 4.2) tells Knockout to bind the properties of the ViewModel object to an attribute of the containing element (in this case the value attribute of the input element). There is a large set of predefined bindings for different purposes. There are controlling and appearance bindings such as the visible or the style binding, control flow bindings such as the foreach binding to iterate over a set of ViewModel properties and at least, form field bindings. If these bindings are not sufficient it is possible to create custom bindings.

The last line in listing 4.1 binds the ViewModel to the view. After that all changes in the ViewModel are reflected to the input element and vice versa. The ViewModel itself defines two different sort of properties. The properties 'firstName' and 'lastName' are normal observables. The 'fullName' property is a computed observable which can be any calculated value, also considering other observables. Since the properties are special Knockout objects, it is necessary to access the property values like a function (see line 5 in listing 4.1).

Knockout also provides a more convenient way to define the properties of a ViewModel, the mapping plugin. The mapping plugin converts any property of any JSON object to a Knockout observable in order to act as ViewModel. The following listing (4.3) shows the above introduced name example with the mapping plugin.

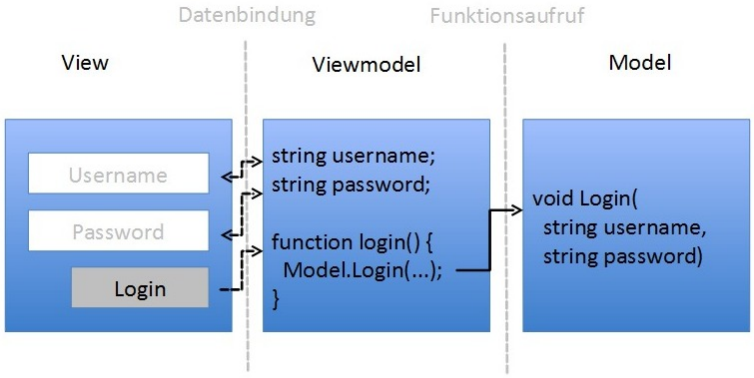


Figure 4.10: MVVM pattern, source [15]

Listing 4.3: the mapping plugin

```
1 var person = {
2   firstName : 'Daniel';
3   lastName : 'Meiers';
4   fullname : function({
5       return firstname +lastName;
6     });
7 var viewModel = ko.mapping.fromJS(person);
```

Knockout JS doesn't support attaching a server side backend or REST API. Furthermore it is not able to route between different pages of your application. Thus these parts of the SPA has to be implemented by your own or with additional frameworks.

Ember JS

Ember JS is one of the younger frameworks, mainly created by Yehuda Katz and Tom Dale and introduced in December 2011 [42]. In this short time Ember JS could make a really impressive progress and has gained a lot of attraction which can be concluded from the number of github stars and watches. It comprises all necessary parts for building web applications including two way data binding with templates, a routing mechanism and provides a way to connect the application to REST API's.

Ember is a very stringent and opinionated framework and is built upon concepts such as DRY (dont repeat yourself) and CoC (Convention over Configuration). Especially the usage of naming conventions allows it to write applications with a nominal amount of code. The disadvantage of this is, that it is more sophisticated to get started with Ember.js because the philosophy and the concepts of Ember need to be understand in first step. The full stack and opinionated approach are the reason why Ember is one of the largest frameworks (56k).

It uses Handlebars as its templating engine, which is a very popular string based templating engine. Ember.js also provides a way to connect your application to RESTful WebServices with an additional project ember-data. As well as Ember itself, ember-data highly makes use of the Convention over Configuration principle which means that ember-data expects the data provided by a REST-Service in a special format. The Ember team substantiate : "[...]we don't think most web developers should have to write any custom XHR code for loading data. Strong conventions on the client and strong conventions on the server should allow them to communicate automatically." [43]. Besides Ember-Data it is also possible to use any other server connection implementation.

To build an Ember application it is first necessary to create an instance of an Ember.Application object (see listing 4.4).

Listing 4.4: app.js

```
1 var App = Ember.Application.create();
```

Listing 4.5: index.html

```
1 <html>
2     <head>
3         <title></title>
4         <meta http-equiv="Content-Type" content="text/html;
              charset=UTF-8">
5     </head>
6     <body>
7
8     </body>
9
10    <script type="text/x-handlebars">
```

```
11   <div>
12   {{outlet}}
13   </div>
14     </script>
15
16     <script type="text/x-handlebars" data-template-name="index
         ">
17   <div>
18     <label>First Name</label>
19     <input {{bindAttr value=firstName}}></input>
20   </div>
21   <div>
22     <label>Last Name</label>
23     <input {{bindAttr value=LastName}}></input>
24   </div>
25   <p>Hello {{fullName}}</p>
26    </script>
27
28     <script src="js/libs/jquery-1.9.1.js"></script>
29     <script src="js/libs/handlebars-1.0.0-rc.4.js"></script>
30     <script src="js/libs/ember-1.0.0-rc.6.js"></script>
31     <script src="js/app.js"></script>
32 </html>
```

Creating the application object defines the namespace of the application. All other classes, like Routers, Controllers, Views or Models are defined as properties of the application object. This has the advantage that the JavaScript namespace does not get polluted by application depended objects. Besides some other initialisation, the create() statement automatically creates a default router for the application. The default router first sets up the ApplicationRoute with the ApplicationTemplate, a pre-defined and special route only for application startup.The ApplicationTemplate is defined in listing 4.5 in line 10. By convention Ember treats this as Application template in fact the id attribute of the script tag is missing.

The ApplicationTemplate is the right place for static content like headers, footers or menu bars, in fact it gets always rendered first. The {{outlet}} tag tells ember where to fill in dynamic content that depends on the currently loaded route. After

URL	Route Name	Controller	Route	Template
/	index	IndexController	IndexRoute	index
/about	about	AboutController	AboutRoute	about
/favs	favorites	FavoritesController	FavoritesRoute	favorites

Figure 4.11: Ember Conventions for Routes, source [44]

this initialization the Ember router starts the routing process and routes to the current route. This is normally the plain basic url. This path is connected per convetion to Embers IndexRoute so that the Router takes all steps to initialize the IndexRoute. One important note to mention here, is that the templates are standard handlebars templates.

As outlined above, a very central concept of Ember is routing. Every Ember application needs a Router in fact the router translates a URL into a series of templates and is responsible for loading these templates as well as respective model data and sets up other application state. Listing 4.6 demonstrates a Router definition with two different routes.

Listing 4.6: Router Defintion in Ember

```
1 App.Router.map(function() {
2   this.route("about");
3   this.route("favorites");
4 });
```

Embers routing mechanism is good example how the CoC apporach is used in Ember to avoid boilerplate code. As listing 4.6 depicts, only the routes itself are defined but no programming logic that defines how to set up the routes. This is because a naming convention is used to reason from the current url to the route name and from the route name to the controller and the template to display. The table in figure 4.11 demonstrates the naming convention.

Starting the above mentioned code snippets would result in a page showing empty input fields for both properties, because the templates that are involved are not backed with a model that contains data the templates should display.

Every template is backed by a model, and the route defines what model should be used by the template . As depicted in the table above, ember will look for an App.IndexRoute by convention, so this is the right place where we need to point to the model.

Listing 4.7: app.js

```
1 var App = Ember.Application.create();
2 App.Person = Ember.Object.extend({
3   firstName: null,
4   lastName : null,
5     fullName: function() {
6         return this.get('firstName') + " " + this.get('
              lastName');
7     }.property('firstName','lastName')
8 });
9
10 var myself = App.Person.create({
11     firstName: "Daniel",
12     lastName: "Meiers",
13 });
14
15 App.IndexController = Ember.ObjectController.extend({});
16
17 App.IndexRoute = Ember.Route.extend({
18     model: function() {
19         return myself;
20     }
21 });
```

Listing 4.7 needs some more explanation. First at all we define a model class called Person which has the same properties as in the Knockout example. In order to do this, we need to extend a special ember class in fact we want to bind to these properties int the templates. After that, we create an instance of this class and set the needed Properties. In line 15 we declare the controller which should be used for the IndexRoute. In Ember a Controller is the component that manages the display logic and the application state. It is comparable to Knockout's ViewModel object. As mentioned earlier, each template is backed by a model. To be more precise, templates retrieve their properties from the

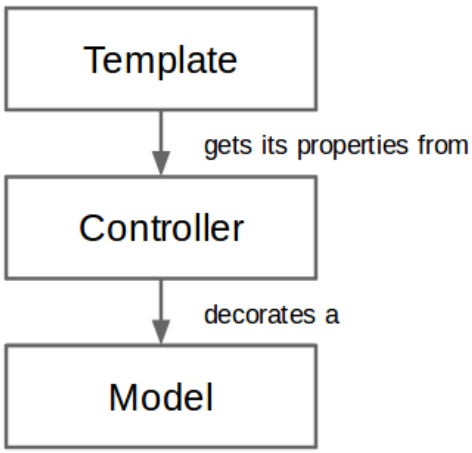


Figure 4.12: Ember model coupling, source [45]

controller which decorates the model and provide proxy attributes to avoid for an easier access in the templates. Figure 4.12 visualise how templates, controllers and model are related to each other. In the IndexRoute object, we finally point to our model instance.

Per default it is not necessary to define a controller for the IndexRoute, but by convention Ember assumes that the IndexController manages a list of model objects and therefore expects an array for the model property. Another way to achieve the same thing, is demonstrated in the following listing. We simply define an special Ember Array and add our created Person instance to it, and finally point the IndexRoute to the array instead. Since we now have an array as our model, we need to adjust the template, to iterate over all existing array members (see listing 4.9).

Listing 4.8: app.js

```
1 var persons = Ember.A();
2 persons.pushObject(myself);
3
4 App.IndexRoute = Ember.Route.extend({
5     model: function() {
6         return myself;
7     }
8 });
```

Listing 4.9: index.html

```
<script type="text/x-handlebars" data-template-name="index">
  {{#each}}
    <div>
      <label>First Name</label>
      <input {{bindAttr value=firstName}}></input>
    </div>
    <div>
      <label>Last Name</label>
      <input {{bindAttr value=LastName}}></input>
    </div>
    <p>{{fullName}}</p>
  {{/each}}
</script>
```

If we start that application we see that the input fields are properly bound to the model properties. But if we change the first name or last name in the input field these changes are not reflected to the p-element at the bottom of the page. This is because Ember does not know how to react on events that are fired on user reaction. If we want to react on user events and enable a two way data binding, we need to use an Ember View instead. In Ember, views "[...] are responsible for responding to user events, like clicks, drags, and scrolls, as well as updating the contents of the DOM when the data underlying the view changes." [46]. Luckily, Ember has a small set of built in views which are Ember.Checkbox, Ember.TextField, Ember.Select and Ember.Textarea. If necessary, it is also possible to create custom views. Therefore the only thing we need to change is to replace the input element with an Ember.TextField view as demonstrated in listing 4.10

Listing 4.10: Ember Views

```
  {{view Ember.TextField valueBinding="firstName"}}
```

As mentioned earlier, Ember also provides a way to automatically connect Ember Models to an REST API. However, an explanation of this module would be out of the scope of this chapter, hence no furhter basic concepts are involved. Fur-

thermore, using ember data has some crucial disadvantages which are discussed in section 4.2.3.

Angular JS

AngularJS is developed by Google and has become one of the more popular Frameworks. AngularJS follows a slightly different apporach compared to the other frameworks. This approach is best described by the Angular developers itself: "Angular is what HTML would have been had it been designed for applications" [47]. One aspect that shows the difference of Angular are the so called directives, which are a key concept of AngularJS. Directives allow the user to extend the native HTML with new and application dependent functionality. This is done by introducing new HTML tags or attributes, as well as some JavaScript code that defines the behaviour and a way how to convert the new introduced tag and the current model state into plain old HTML. Using directives extensively, HTML can be turned into a declarative domain specific language (DSL).

Furthermore AngularJS also provides two way data binding, filters, routing and dependency injection. All these components and features make it easy to write Single Page Web Applications without writing any sort of boilerplate code, assuming the architecture and concepts of Angular JS are well known.

Listing 4.11 shows the very simple implementation of the same app used before. The special here is that it is not necessary to write one line of JavaScript code to get things work. Unfortunately this example isn't very intuitive and it seems that there happens a lot of magic behind the scenes. In the following sections the details of the example and the important concepts of AngularJS are explained.

Listing 4.11: First Angular application

```
1 <html ng-app>
2     <head>
3         <meta charset="utf-8">
4         <title>My AngularJS App</title>
5     </head>
6     <body ng-init="firstName='Daniel'; lastName='Meiers';">
7         <div>
```

```
8               <label>First Name</label>
9               <input ng-model="firstName"></input>
10          </div>
11          <div>
12              <label>Last Name</label>
13              <input ng-Model="lastName"></input>
14          </div>
15          <p>Hello {{firstName}} {{lastName}}</p>
16          <script src="lib/angular/angular.js"></script>
17          <script src="js/app.js"></script>
18      </body>
19  </html>
```

The first thing to mention here is the ng-app directive. This directive defines the application root and Angular automatically looks for this directive during page load. If found, it starts bootstrapping the angular application which basically means three things:

- load the module associated with the directive.
- create the application injector
- compile the DOM treating the ng-app directive as the root of the compilation.

Using this special directive as the root of the Angular related content has the advantage that only a part of the total application can be handled by Angular and the context of the Angular application can be restricted to a part of the DOM. This makes it much easier to migrate Angular into existing projects or to combine Angular with other frameworks.

In Angular the application consists of one or more so called Modules. Modules "[...] declaratively specify how an application should be bootstrapped" [48]. This is necessary since Angular applications does not have a main method which can make the instantiation process. The Module that should be loaded can be referenced in the ng-app directive.

Listing 4.12: ng-app declaration with module

```
1 <html ng-app="myApp">
2 ....
3 <script type="text">
4   var myAppModule = angular.module('myApp', []);
5 </script>
6 ....
```

The first parameter of the module creation (line 4 of listing 4.12) defines the name of the module that must be used for referencing it in the ng-app directive. The second parameter can be used to define other modules as dependency for this module. These dependent modules are created first and then injected into the module per Dependency Injection. Among others,the most notably advantage of this approach is, that the separation in modules simplifies unit testing, since not all modules must be loaded for unit testing and additional modules can be loaded, that, can provide some mock up functionality for testing. This helps writing end-to-end test for the application and allows easily testing the GUI behaviour with unit tests.

In our simple application (see listing 4.11) we have not defined a module that should be loaded. Therefore angular loads a default module. This default module initializes the application wide Injector which is responsible for the Dependency Injection. The last step of bootstrapping comprises the re-compilation of the DOM. Each found Directive or binding expression is evaluated and translated into plain HTML.

The next directive we used in 4.11 is the ng-init directive. this directive allows to do initialisation tasks before the app starts running. The ng-init directive automatically creates a new property on the corresponding scope object and sets the respective initial values for it, that are declared in the expression. A more detailed explanation of the scope concept is given later on, but in short, the scope in Angular can be compared with the ViewModel object in Knockout. The ng-model directive binds those properties to the element it is defined in, which automatically enables a two-way data binding between the model properties and the view. The automatic creation of model properties, is an important difference

Figure 4.13: Hirarchie of Angular scopes, source [49]

compared to Knockout and Ember, where this needs to be done manually. Before explaining this feature in more detail it is necessary to have a closer look on Angualars scope concept which also requires to understand the meaning of Models, Views and Controllers in the Angular architecture.

The Scope is an object that refers to the application model and provides necessary context for Expressions and directives. An Angular expressions are a JavaScript-like code snippets that are usually places in data-bindings and can be used for accessing model properties and some basic calculations. Scopes can watch Angular expressions and propagate events. These features are used also internally to implement the two way data binding. Scopes are the "glue between application controller and the view" [49] and are comparable to Knockouts ViewModel. An Angular application always consists of exactly one RootScope and multiple ChildScopes. They are hierarchical nested and resemble the DOM structure as figure 4.13 depicts. Similar to the model declaration, this can also happen implicit by directives like the ng-repeat directive in figure 4.13. ChildScopes prototypically inherit from their parent scope. Hence, if an binding expression like firstName is evaluated, Angular first checks if the Scope object related to that element contains this property. If not it checks all parent scopes until the property is found or the RootScope is reached.

In Angular a Model can be any JavaScript object including Arrays and primitives.

The only condition is that it must be referenced by a Scope object. The name of the scope property is also the model identifier that can be used for accessing the value in expressions. The creation of models can be done explicit or implicit. In listing 4.11 the model is created implicit with the ng-model directive. To create a model explicit it is necessary to add new properties to the scope in the corresponding Controller.

As already mentioned, Controllers are also a way to create models in Angular and adding them to the Scope. Controllers are JavaScript functions that are used to augment the Angular scope and are normally used to set up initial state of the Scope or to add application behaviour to it. Listing 4.13 depicts the needed changes.

Listing 4.13: Angular Controller

```
1 <html>
2 ....
3 <body ng-controller="HelloController">
4 ....
5 <script type="text">
6   function HelloController($scope){
7       $scope.lastName = "Meiers";
8       $scope.firstName = "Daniel";
9   };
10 </script>
11 ...
```

There are two ways how Controllers are associated with Scope objects. The first one is to use the ng-controller directive as in the example above. The second one is to use Angulars Routing mechanism. Another important thing to mention here is that the Scope object is provided by Angulars DI (Dependenci Injection) system.

The last component to cover all elements of the MVC pattern is the View. In Angular the view consists of the loaded and rendered DOM after Angular has transformed Angular specific directives and expressions into normal HTML. Figure 4.14 are giving a good overview over this process.

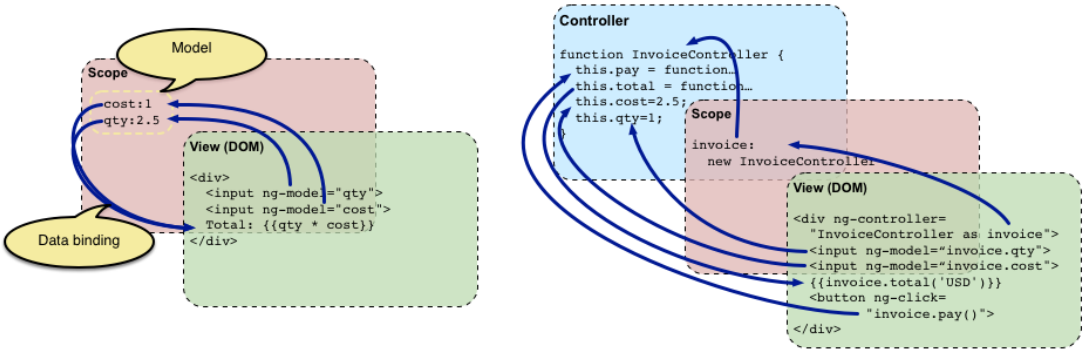


Figure 4.14: Functionality of Angular Views, source [50]

There are a few more concepts in Angular that helps and simplifies SPA development with Angular. A really useful feature that makes Angular outstanding are Filters. Filters are a way to format data that should be displayed to the user. For example, if we want to print the fullName property of our basic example in uppercase letters we just need to use the Uppercase filter provided by Angular (see listing 4.14). Angular already provides a basic set of very useful Filters among them Filters for ordering or filtering lists of data but it is also possible to create custom ones. Filters are used in Angular expressions with an pipe like syntax.

Listing 4.14: Angular Filters

```
1 {{ fullName | uppercase}}
```

The last concept are Services. Services in Angular apps are substitutable objects that are wired together using dependency injection. Angular Services are implemented as singletons and are used to carry out specific tasks common to web apps and reusable business logic independent of views. Angular provides an impressive set of services such as the $http service that provides low level access to the browser's XMLHttpRequest object and can be used to make AJAX requests.

The routing mechanism is also implemented as service. This has the advantage that it can be easily injected by dependency injection in every module that wants to use the routing mechanism. The oruting service tries to map the path of the current location to an existing route definition. Listing 4.15 demonstrates that a route definition basically consists of the template that should be loaded and the controller function the route is related with. The url the route is defined for

can also contain dynamic parts, for example the bookId field, that indicates the controller which book object shall be displayed.

Listing 4.15: Route definition in AngularJS

```
var mod = angular.module('ngViewExample', ['ngRoute'])

mod.config(function($routeProvider, $locationProvider) {
  $routeProvider.when('/Book/:bookId', {
    templateUrl: 'book.html',
    controller: BookCntl
    }
  });
  $routeProvider.when('/Book/:bookId/ch/:chapterId', {
    templateUrl: 'chapter.html',
    controller: ChapterCntl
  });

  function BookCntl($scope, $routeParams) {
  ....
  }
});
```

4.2.3 Discussion

After giving a first introduction into the concepts of Knockout.js, Ember.js and AngularJS, it is necessary to discuss the pros and cons of the different frameworks.

Knockout JS very reduced feature set can be both, a blessing and a curse and it depends on the needs to decide what of these two things prevails. Knockout JS is a very easy to learn library hence there are only a few concepts to learn. Another benefit of Knockout JS is, that it has a widely spreaded browser support also for older browser versions. This benefit is not that much relevant since the applications to develop should be rendered in the JavaFX WebKit component. Knockout is a very flexible framework hence there are no regulations for defining the structure and architecture of your application. The missing features of Knockout JS

like routing can be compensated with one of the other dozen available third party libraries for this feature. Unfortunately, without these additional dependencies Knockout JS is rather less suited for building more complex web applications in fact that it forces you to implement too many things manually. And the additional amount of dependencies could lead to inconsistency problems, increases the error likelihood and the maintenance efforts and make the development even harder (cf [15]).

Ember however, covers all needed aspects of developing SPA's. It is possible to write applications with a minimal amount of code, and its concept scale very well, which means it is very easy to write simple applications with Ember that reduces the amount of boilerplate code to a minimum, but the underlying concepts cover also the possibility to create large and complex Single Page Web Applications. With the concept of Ember Components, it is possible to develop reusable components that are furthermore planned to support the upcoming web components standard. The only disadvantage is, that Ember is a very strict and opinionated framework. This is especially true for ember-data which relies heavily on naming conventions to properly work. Although using approaches like Convention over Configuration is not a principal disadvantage, the question is, if the very stringent corset of Ember will limit further developments or produces a lot of efforts to work around the built in framework concepts. Ember-data is a good example for that. Ember-data makes highly use of naming conventions and makes assumptions on the structure of the REST API of the backend. Hence the RESTful API is already defined, it is very likely that the assumptions of ember-data will not correlate with cids REST API, which means that the concepts of the framework need to be work around. It can not be ruled that similar problems could also arise in other parts of the framework, for example the routing mechanism.

At this point, the concepts of dependency injection, and services, makes Angular much more flexible. Similar to Ember, Angular is also capable of building all kind of applications, from the simple to the complex one. From special interest is the very central concept of directives, which can be used to easily develop reusable components. The angular developers claim, that they will support compatibility of directives to the upcoming web component standard. The large number of concepts makes it a bit harder to get familiar with Angular, but if the concepts

are clear, it is very intuitive to build applications with Angular. Outstanding points of Angular are besides directives, the usage of Dependency Injection, which allows to easily reuse developed Filters, Services and Modules. Additionally the dependency injection increases the testability of the applications and gives the possibility to easily write end to end unit tests.

Regarding the performance of Angular and Ember there are some important notes to mention. As outlined in chapter 2 Angular and Ember follow two very different approaches for achieving data binding. In Ember it is necessary to use special objects for the model. Those objects makes it possible to notify about changes. In Angular it is possible to bind to arbitrary JavaScript objects. It uses dirty-checking to test if any of the binded objects has changed. The two approaches have some fundamental consequences. The first and most obvious one is that using dirty checking means that it is possible to bind directly to native JavaScript objects. This makes the binding expressions slightly more intuitive since no get or set functions are needed to access the properties. Furthermore it is not necessary to wrap object into framework depended observable object. On the other side, the general approach to periodically check the bound JavaScript objects for changes seems to be time consuming and slow. Although this is conceptually true, this statement needs to be qualified in the practical context. A good discussion regarding this issue can be found at [51]. The main arguments mentioned there are, that dirty checking is only problematic for a large number of objects and that the human eye can not recognize changes faster than a 50 ms. There need to be a large amount of objects to check before the 50 ms are exceeded and the user can recognize a sluggish UI.

In addition, also Ember has performance bottlenecks. In [17], Marius Grundersen reveals and discusses the weaknesses of Ember and Angular and demonstrates them with a simple example. Figure 4.15 demonstrates the structure of the demo application. It mainly conists of a input fiels which binds the inserted text to a label above. Furthermore, multiple thousands items are added by clicking a button.

Executing this example in Ember reveals that it takes a considerably time until all list items are rendered. Figure 4.16 demonstrates the time that is needed for this example. Important to note is, that Angular is much faster in that case

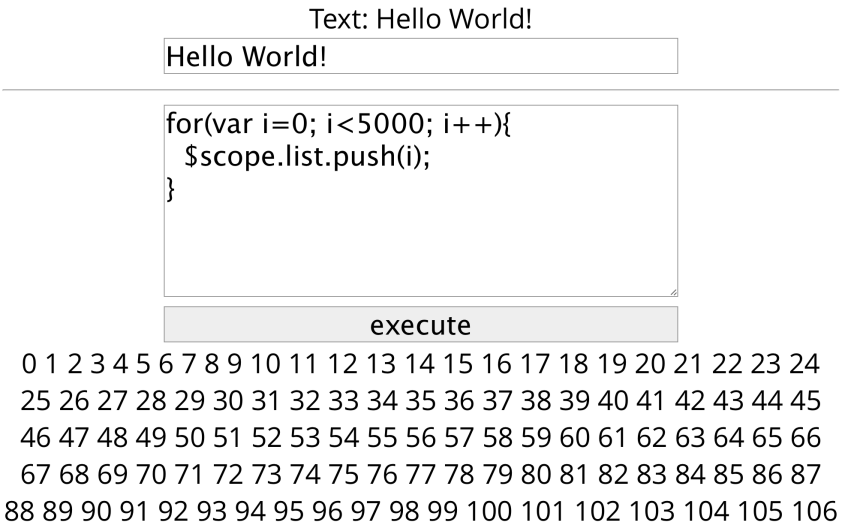


Figure 4.15: Demo Application, source [17]

despite the dirty checking approach. The reason for this is, that in Ember it is necessary to set up the notification mechanism for the 5000 objects. To be fair, the disadvantage of dirty checking first appears when typing in the text field after the list is rendered. In this case Angular needs to check all 5000 items although only 1 has changed. This makes typing a bit sluggish. This example impressively demonstrates that both approaches can have bottleneck effects and it depends on the developer to regard the performance when developing the application. A simple solution to avoid the bottleneck affect in Angular is two use multiple scopes.

Another important part that is also listed in the requirements in chapter 3.3 is the documentation, the community support and the existence of test tools. But also here the difference between Ember and Angular are more from a marginal nature. The documentations are both very extensive and from a good quality. There a many examples that allow to easily use the frameworks. Furthermore the documentaion of Ember as well the Angular documentation both provide tutorial ans How-To articles. However, from a subjective point of view, it appears the the Angular documention has a slight advantage, which may be due to the fact that Ember is the much younger framework. The maturity of Angular can also be the reason for the larger community and popularity of the framework. But again, for bot frameworks it is easy to find information or get help.

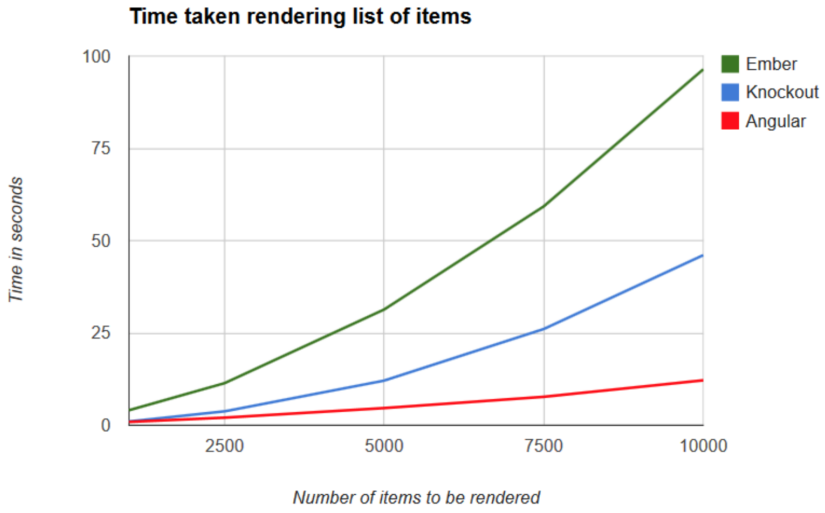


Figure 4.16: Performance Comparison of Demo Application, source [17]

Regarding the support of test and debugging tools it seems that Angular has a slight advantage hence it has a very good built in testability and also provides a debugging tool Batarang. Batarang helps to examine the scope structure the application and helps monitoring and optimzing the performance. Unfortunately Batarang is only available as Google Chrome extension and cannot be used in the JavaFX WebView.

Summing up, Knockout.js can not be used, hence it has a reduced feature set and does not offer a routing mechanism or reusable components. Embers very stringent and opinionated approach bears the danger that the framework can limit future developments and complicated and sophisticated workarounds are necessary. Hence AngularJS has none of the above mentioned disadvantages, it is clear that AngularJS is the most promising candidate and shall be used for the further considerations. Table 4.2.3 gives an overview over the different features of each framework and allows a more accessible comparison of them. The disqualifying factors are emphasized with with a coloured background. Up to know the conception of the new required feature is outlined, and the most suiting candidate regarding the browser API as well as a JavaScript MVC Framework is evaluated. Now, that all prerequisites are fulfilled it is possible to start with the implementation.

	Angular JS	Ember JS	Knockout JS
2-way data binding	✓	✓	✓
normal and computed properties	✓/ change detection	✓/ ✓	✓/ ✓
Testing	end-to-end tests, build in tools for unit tests,debugging extension for chrome	extra packages ember-testing,qUnit	unit tests
Routing	✓	✓	-
Multiple and Composite Views	✓/ ✓	✓/ ✓	-
Backend connection	optional (ajax wrapper)	optional (ember-data)	-
Documentation	very good	good	good
API stability / maturity	since 2009, stable	since 2011,	since 2010, very stable
community (github watch/star/-fork	1450/13061/3147	672/7752/1568	349/4074/657
Perfomance	slowest when the model is complex	slowest when rendering large lists	slowest when pushing many items
Flexibile vs opinonated	flexible	stringent & opinionated	flexible
Integration of UI Elements	✓use directives	✓use Ember views	✓use custom bindings
License	MIT License	MIT License	MIT License

5 Implementation

5.1 Integrating the JavaFX WebView

5.1.1 JavaFX WebView based DescriptionPane

As a first step it is necessary to integrate the JavaFX WebView into the description pane of the navigator. As already mentioned the description pane already supports the visualisation of HTML pages to offer additional information for the selected catalogue object. Hence the JavaFX WebView offers much more features than the already used browser APIs (cf 4.1), a useful benefit can be reached using the WebView as browser API also for that case. To ensure a full backward compatibility the usage of the JavaFX WebView is implemented in a configurable manner. The description pane is mainly implemented as an abstract class `DescriptionPane`. This class represents a `JPanel` and offer basic functionality like the distinction when to load a HTML page or a cids renderer and the loading mechanism for cids renderer. Using a abstract class easily allows to extend the functionality of the `DescriptionPane` that uses different browser API's. Figure 5.1 depicts a class diagram of the `DescriptionPane` and related classes.

The `DescriptionPane` panel uses a `CardLayout` to separate the visualisation of cids renderer from the visualisation of HTML pages. The `CardLayout` easily allows to programmatically switch between multiple and different panels based on unique names, without taking care of refreshing the GUI by hand. Using this approach the implementation classes of the `DescriptionPane` can easily implement their own logic and define the tools they use to visualise HTML description pages. It is just necessary to add a panel to the `DescriptionPane` with the correct identifier that is used in the abstract class. The three abstract methods

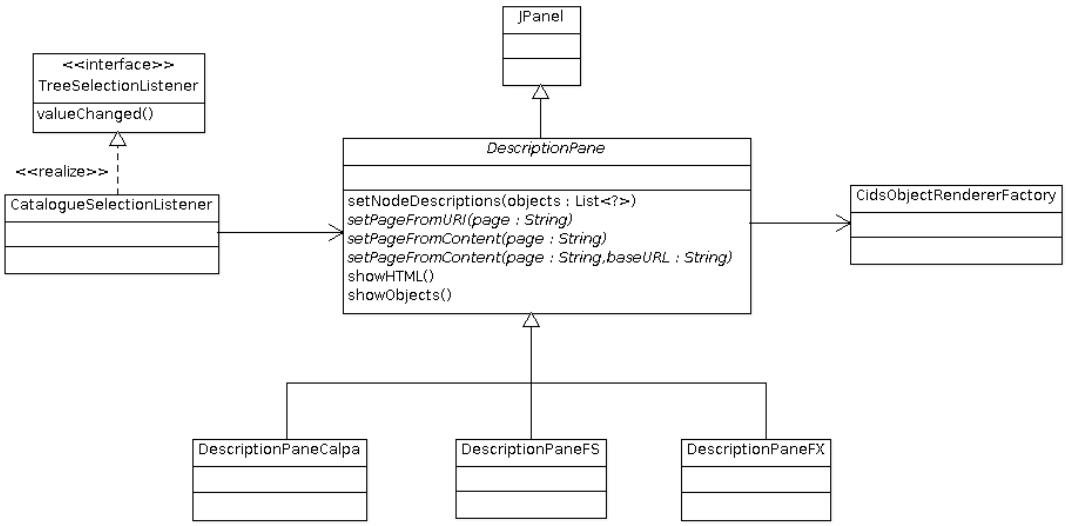


Figure 5.1: DescriptionPane and relating classes

shown in figure 5.1, are used to provide the HTML encoded information that shall be displayed.

Which concrete implementation of the `DescriptionPane` is used, is determined during the navigator start and depends on the configuration made in the navigators configuration file. During the navigator start, the `PropertyManager` class loads the configuration file of the navigator and caches the settings. For each property the `PropertyManager` has corresponding getter and setter functions. It is implemented as a singleton, which easily allows to use it everywhere where access to the settings is needed. There are already two existing implementations that uses different browser APIs. Hence the current implementation uses Calpa API as default case, there is only one boolean property called `useFlyingSaucer` that indicates to use the `DescriptionPaneFS` class, a description pane that uses the FlyingSaucer API. One possible solution for introducing a JavaFX based description pane is to introduce a new property similar to the already existing one. Following this approach has the drawback that there are different properties for configuring the same functionality. Furthermore, the name of the "`useFlyingSaucer`" property does not indicate very well what functionality is exactly configured. Therefore a new text property "`navigator.descriptionPane.htmlRenderer`" is introduced, whose value determines which concrete implementation of the

`DescriptionPane` shall be used. To ensure backward compatibility the old properties are still supported, but when they are used a warning is written to the logging mechanism of the application. Additionally, the `DescriptionPaneCalpa` still functions as fall-back case if no or a invalid value for the property is configured.

For the JavaFX WebView based description pane, a new class `DescriptionPaneFX` is generated, that uses a `JFXPanel` that contains a Java FX WebView for visualising HTML content. This panel is then used within the `CardLayout` of the description pane as described above. The `JFXPanel` is a Swing component that acts as JavaFX application runtime container and eases the integration of JavaFX components into Swing. Normally when developing JavaFX applications, it is necessary to extend the JavaFX `Application` class, which initializes the JavaFX runtime environment and automatically calls a `start(Stage primaryStage)` method that needs to be implemented when extending the `Application` class. Important to note is the parameter of this method. A stage object is a top level GUI container that can have several scenes. The scenes itself are also container classes for contents. This architecture is based on the principal idea of a theatre play. The stage on a theatre contains multiple scenes, each with a custom look and behaviour. The stage object is already constructed by the `Application` class and it is just necessary to set a scene object on that stage. When using a `JFXPanel`, there is no start method that can be executed. Therefore the `JFXPanel` initializes the JavaFX runtime environment during construction time. The only thing what needs to be done is to set a scene that should be visualised by the `JFXPanel` which in our case is the WebView. Figure 5.2 demonstrates the class structure of the FX based description pane.

Using JavaFX dependent classes requires to integrate the JavaFX library into the classpath. Although this library is shipped with the JRE since Java SE 7 update 6 (7u6), it is not part of the default classpath that is used when building and starting Netbeans projects. Thus, it is not possible to use and built programs that use JavaFX dependent classes within the Netbeans IDE. Possible solutions to this issue are discussed in [52]. The first approach is to move the corresponding jar file into a folder that is on the classpath by default, like the ext-folder of the JRE. This has the disadvantage that this adoption needs to be done on every client.

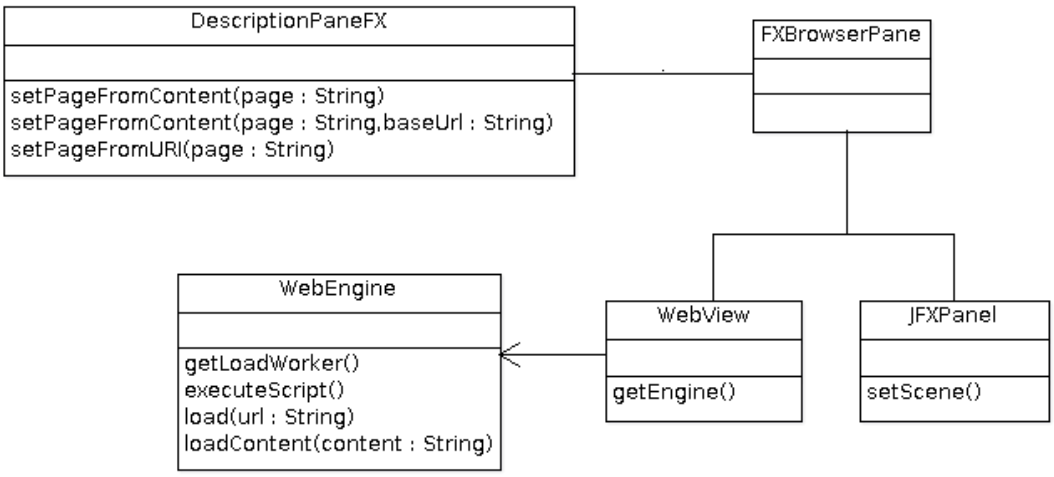


Figure 5.2: DescriptionPaneFX and relating classes

Therefore an other solution is preferred. The jfxrt.jar gets defined as system dependency. System dependencies are always available and are not looked up in the maven specific repository, hence maven assumes that these dependencies are explicitly provided. Thus way, the JavaFX dependency can be defined in a system and JRE independent way. Additionally it is necessary to add the library to the classpath that is created from Maven for running the application and to avoid a `java.lang.NoClassDefFoundError` when running the application.

During the development, two problems occurred. The first one was an `Illegal StateException: Not on FX application thread` Exception. The reason for this is quite simple and is based on the fact, that JavaFX components can not be accessed outside the JavaFX application thread. This is similar to JavaSwing where Swing components should also be accessed only from the Event Dispatch Thread, with the difference that violating this constraint leads to an exception when trying the same in JavaFX. This was a conscious decision from the JavaFX developers hence accessing Swing components outside the Event Dispatch Thread is one of the most reason for errors.

The second problem occurred, when switching between a object node that nudges the description pane to load and show a Swing based cids renderer and a node with an HTML description page. When switching back to the HTML description again an `IllegalStateException` is thrown, with the message `"Platform exit`

`has been called"`. The reason for this is the implementation of the `JFXPanel`. Hence the `JFXPanel` automatically starts the JavaFX runtime environment, it must also ensure it is terminated when it is no longer necessary. This is the case when the scene object that the `JFXPanel` displays is finished and the panel is no longer visible. Hence we use the `JFXPanel`in a `CardLayout` the `JFXPanel` implicitly calls the exit method of the JavaFX platform which terminates the JavaFX environment. This implicit termination can be suppressed by setting a parameter with the following code:

Listing 5.1: Disable JavaFX implicit exit

```
1 Platform.setImplicitExit(false);
```

5.1.2 Adopting the Convention over Configuration Mechanism

Up to this point the JavaFX WebView is already integrated into the navigator, but it is still not possible to visualize html applications as renderer for object nodes. As mentioned above, the abstract class `DescriptionPane` is responsible for loading the corresponding renderer for object nodes. To fully understand what possibilities exists to implement the pending feature, it is necessary to have a closer look on that mechanism. When selecting a node in the cids catalogue, the `CatalogueSelectionListener` gets notified about this. The `CatalogueSelectionListener` calls the `setNodeDescription` method on the `DescriptionPane` This method makes a series of decisions what exactly should be displayed.

The first idea to show HTML applications as cids renderer is to customize this method and implement the necessary changes there. Since there can be different description pane implementations, but only the `DescriptionPaneFX` can display complex web applications it is necessary to overwrite this method there. The code needs to be changed to implement the following logic. If the selected node is a object node, it is necessary to check if for the corresponding cids class the new introduced class attribute `isHtmlRenderer` is configured. If so, the value of the class attribute points to the location of the web application. This URL can be used to call the `setPageFromURL()` method of the `DescriptionPaneFX`.

The drawback of this approach is, that it will only work when the new JavaFX based description pane is used and will not work with the other description pane implementations. Furthermore this approach will not fit perfectly into the conceptual design that strictly separates the display of renderer components and description pages. Therefore a more appropriate solution is to adopt the factories that are responsible for loading the cids renderer and editors for a cids class. As figure 5.1 depicts the `CidsObjectRendererFactory` has two different methods for retrieving single and aggregation renderer. The cids system also allows the definition of so called aggregation renderer which can visualise aggregated information of multiple objects of the same class. If no aggregation renderer is defined, or objects of different classes are selected, the single renderer for each object are vertically stacked in the description pane. Therefore it is just necessary to adopt the `getSingleRenderer` method with the above mentioned changes.

The `CidsObjectRendererFactory` needs to return a `JComponent` that represents the cids renderer. Hence the only difference for web renderer components is the URL that points to the location of the web renderer which is configured in the isHtmlRenderer class attribute, it is easily possible to develop a new renderer class `HTMLWidgetRenderer` that uses also a `FXBrowserPanel`, and is parametrisable with the URL of the web application.

In fact this component can also be used for visualising web based editors, class inheritance can be used to inherit the needed functionality from the editor to the renderer and allows the same code base for both cases. However, this approach heavily relies on the assumption that renderer and editors for single objects have only a different intentional usage and no different graphical user interfaces . A renderer visualizes information whether a editor can be used to edit the data. Using the same code base for editors and renderer is a very common practice that is already used in many cases when developing cids renderer and editors. Using the same GUI for editors and renderer is possible when all input fields of the editor are disabled. Thus, this needs to be regarded when developing web renderer components. Regarding this, the class structure depicted in figure 5.10 is implemented.

Important to note, is the `editable` parameter in the constructor of the `HTMLWidgetEditor`. This parameter can act as a flag if the web renderer component shall be used in

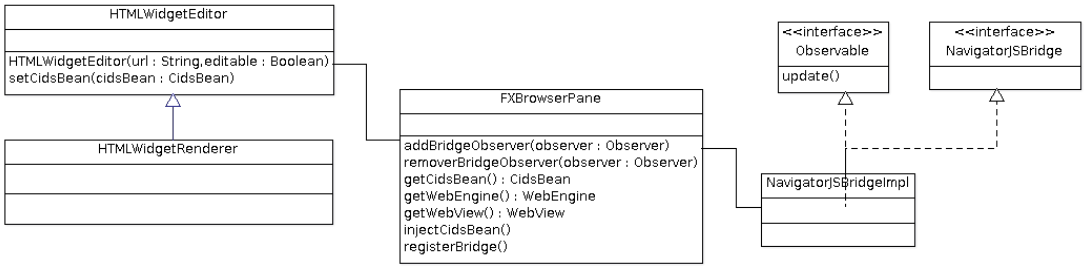


Figure 5.3: HTMLWidgetEditor and related classes

editor or renderer mode, which means that input fields are disabled to avoid data changes. As figure 5.10 depicts, the `HTMLWidgetEditor` implements the `setCidsBean` method, which is called when the renderers and editors are loaded, to inject the object into the renderer. To enable the usage of web applications as cids editors, the only thing left to do is to adopt the `CidsObjectEditorFactory` in the same manner as the `CidsObjectRendererFactory`.

5.2 Development of an Angular based renderer

In a next step a web renderer and editor component is implemented, that facilitates testing the implemented state. Therefore it is necessary to implement an Angular based demo application that can be integrated in the cids navigator as renderer and editor. Without such an application it would be difficult, if not impossible, to test above mentioned enhancements.

For developing a demo application, an already existing render and editor with a complex cids data model shall be used and re-implemented in a web application with the same functionality . In WuNDa, an information system based on cids and the cids navigator, it is possible to manage survey plans. The survey plans are a good candidate for a demo application hence their underlying data structure contains every possible data relation that can be modelled with cids. Figure 5.4 depicts the cids data structure of the survey plans. This data structure is also reflected by the `CidsBean` that need to be injected into the renderer components. The `CidsBean` acts as the main model for the Angular application.

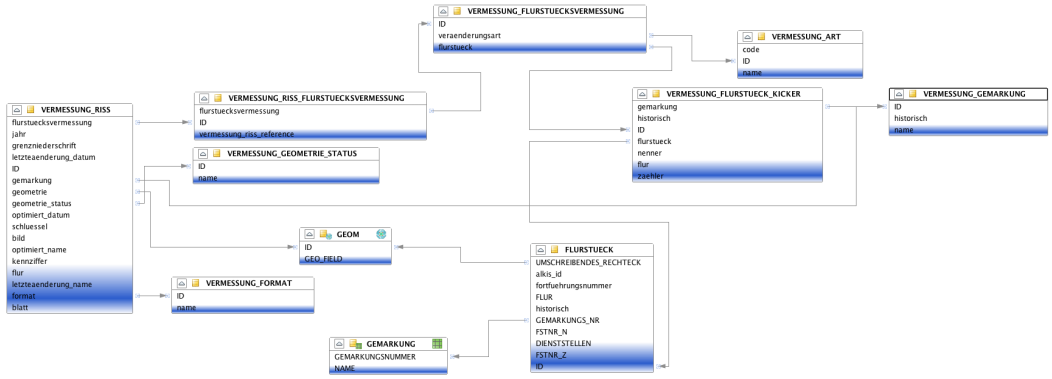


Figure 5.4: Cids data model of survey plans

Before starting with the implementation of the Angular based web renderer, the functionality of the already Swing based implementation of the survey plan renderer and editor is explained. Figure 5.5 shows the Swing based implementation of the survey plan editor. Besides visualising the main information relating to a survey plan (part a in figure 5.5), it is also possible to edit the land parcels that are affected of the survey plan (part b in figure 5.5). Area c in figure 5.5 shows that the survey plan renderer can also visualise images of historic survey plans documents. Thereby, zooming and panning the image is possible. The renderer basically has the same functionality, with the exception that not all information are visualised and it is not possible to change any data. Because implementing the visualisation of historic survey plan documents is more difficult and not mandatory for using and testing the application, this feature is excluded in a first step and can be added later on.

In fact no data exchange between the navigator and the web application is currently possible, it is required to load the data of an example object from a file until the data exchange between navigator and web applications is implemented. According to the usual practice when developing web applications, the data is represented in the JSON format. Hence the objects in the navigator are usually represented as Java `CidsBean` objects, it is necessary to transform and extract the data of a survey plan object into the JSON format. For this the `CidsBean` class offers a set of helper methods that can be used.

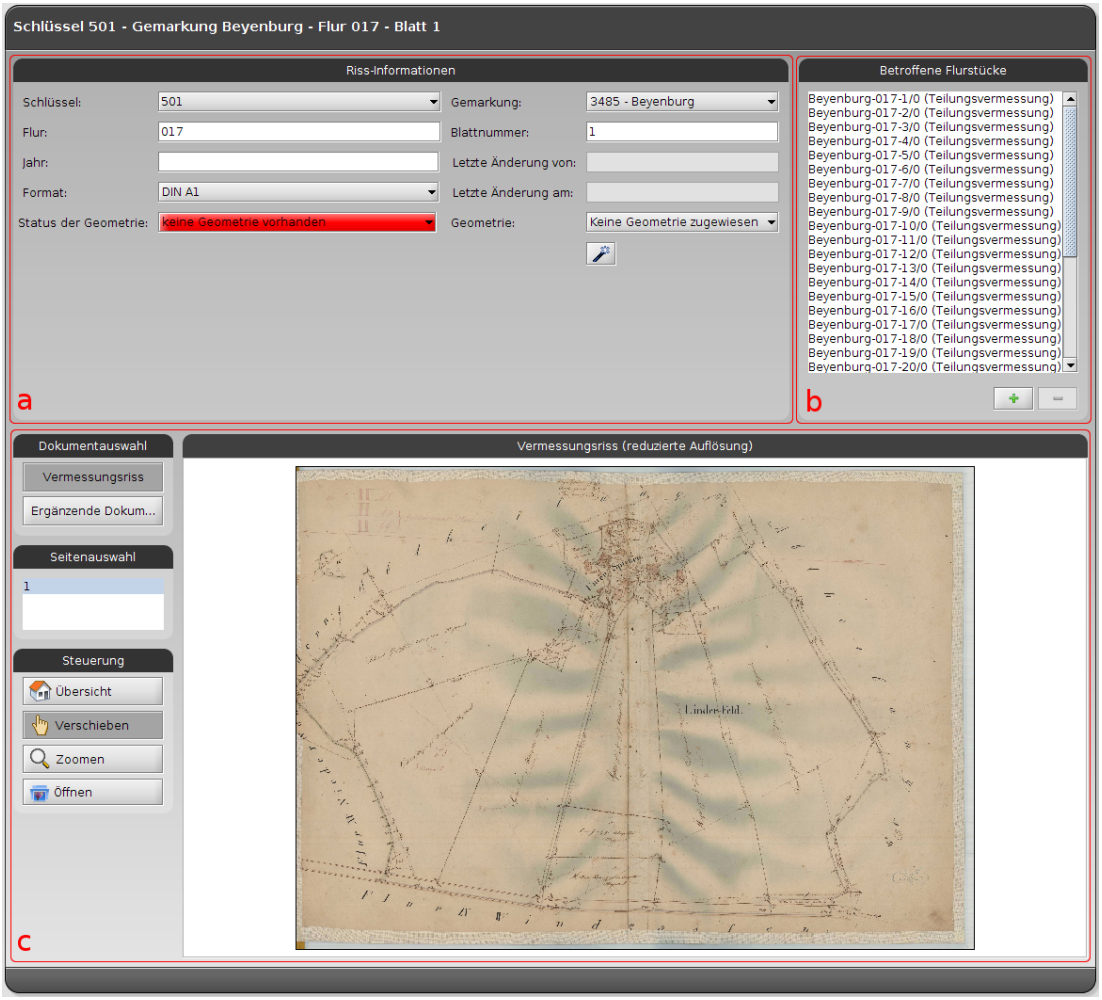


Figure 5.5: Swing based cids editor for survey plans

Listing 5.2: HTML markup of the demo application

```
<!doctype html>
<html lang="en" ng-app="myApp">
 <head>
 </head>
 <body data-ng-controller="SurveyPlanController">
  ...
  <input data-ng-disabled="isRenderer" data-ng-model="cidsBean
     .schluessel"/>
  ...
  <input data-ng-disabled="isRenderer" data-ng-model="cidsBean
     .schluessel"/>
  ...
</body>
</html>
```

Listing 5.3: Controller function of the demo application

```
function SurveyPlanController($scope, $http) {
    $http({method: 'GET', url: './data/surveyPlan.json'}).
            success(function(data, status, headers, config) {
                $scope.cidsBean = data;
            })
}
;
```

A first implementation of the Angular based renderer has a very simple structure. Listing 5.2 and 5.3 represents the important parts of the implementation. For the whole application, one controller with one scope can be used. The controller is defined with the built in directive ng-controller. This directive points Angular, to use the `SurveyPlanController` function as the controller function for the application. The controller loads the JSON formatted representation of the survey plan generated in the previous step. For this, Angular already offers the `$http` service which eases the retrieval of resources in an asynchronous way. When the JSON object is completely loaded, the object can be attached to the `$scope` of the controller. In the view we can easily bind properties of the object to arbitrary elements within the view. The look of the application can be easily adopted with

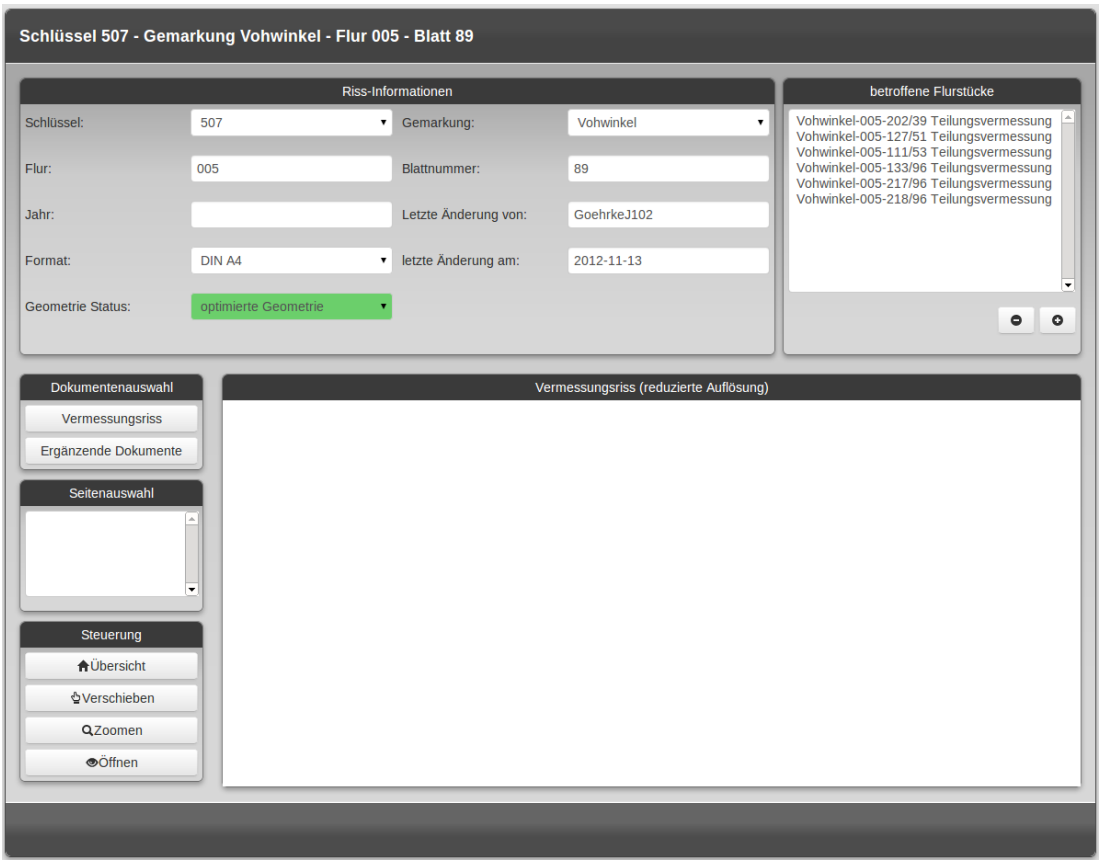


Figure 5.6: Angular based cids editor for survey plans

some additional style sheets to gain a similar design to the already existing swing components. This supports a seamless integration into the cids navigator. Figure 5.6 shows a screenshot of the styled application.

There are still some remaining issues that distinct the web application from the Swing based one. As already mentioned, the same application shall be used as renderer and editor. This means in the case of the demo application that it is necessary to disable the input elements as well as hiding some data fields that are not necessary when the application shall be used as renderer. Therefore a new property called `isRenderer` is added to the scope, that indicates if the application shall be used in renderer or editor mode. This property can be used to hide parts of the view and to change the styling of the input fields. Again, Angular is prepared with built in directives that allow an easy implementation of both features. The ng-hide directive hides arbitrary HTML elements based on the

expression that is provided to it. For disabling the input fields, the `ng-disabled` directive can be used. To both directives it is sufficient to bind the `isRenderer` property to them. When integrating the demo application into the navigator it is necessary to correctly set the `isRenderer` property depending on the usage as renderer or editor. This is described extensively in the next chapter.

Another important difference between the Angular and the Swing based application is how 1-n relations of the data model are visualised to the user. An example is the format field of the survey plan document. A survey plan has a format and there is a fix set of formats that exists. In the Swing application `ComboBoxes` are used to offer the user this fix set of elements of the related cids class.. The `CidsBean` that represents a survey plan only contains just the currently selected element of the related cids class however. In the Swing based application the missing elements are loaded in the background. A same approach needs to be implemented in the Angular based application but requires a working data exchange between the navigator and the web application which is still pending. To ease the integration of this feature afterwards, a Java Script function that loads all elements of a given cids object is implemented. Hence no data exchange with the cids system is possible, those information are also loaded from a json file based on a naming convention. The following examples demonstrates this. For the format example above this method tries to load a file `format.json` from the server. This data can then be used to create select elements that offer all existing format objects. Hence such a select element is a very common and frequently used feature, it is appropriate to implement it as a reusable component. This can be reached with an Angular directive.

The features, adding land parcels to the survey plan, setting a geometry that represents the survey plan and visualising historic survey plan images, all require a working data exchange between the navigator and the web application, or are not essential for using the web application as renderer, or editor respectively, in the navigator. Thus in the next step, the data exchange between the navigator and web applications is implemented first.

5.3 Establishing bi-directional data Exchange

With the current state it is already possible to visualise HTML applications as object renderer and editors within the navigator, but what is still missing, is the data exchange between the navigator and the web applications.

As outlined in chapter 4.1, the JavaFX WebView allows a bi-directional communication between the loaded web page and Java. For the first direction, the communication from Java to the web application, the JavaFX WebEngine has a method `executeScript()`, which allows the execution of arbitrary JavaScript code in the context of the currently loaded page. Furthermore, the WebEngine gives full access to the Document Model (DOM). The DOM can be accessed using Java DOM core classes and allows the manipulation of the DOM programmatically.

Hence Java and JavaScript have a totally different type system, there needs to be a type conversion when exchanging data between Java and JavaScript. In [53] a very detailed explanation of the conversion rules is given. Hence this conversion for simple types is rather obvious, the important part to mention is that JavaScript objects, which can not be easily transferred into simple Java types, are wrapped as an instance of the `JSObject` class. The same conversion is used if the result of the executed JavaScript code is a DOM node.

For making upcalls from JavaScript to Java the concept is to create and register a bridge object, which can be an arbitrary Java object. This bridge object must be accessible in the JavaScript context. This can be reached by calling the `setMember` method on a `JSObject` object. Subsequent, it is possible to access any public fields and methods of the Java bridge object in the JavaScript context. Since the bridge object is registered to a `JSObject`, it is possible to attach the bridge object to any JavaScript object or DOM node. Again a type conversion, that behaves in the inverse way when converting from JavaScript to Java types is applied.

In fact the bridge object can be registered at an arbitrary DOM node or JavaScript object, it is necessary to establish a common interface on which the bridge object always is registered to allow a unified usage within the web application context. Furthermore this interface needs to offer the possibility to inject data. This is

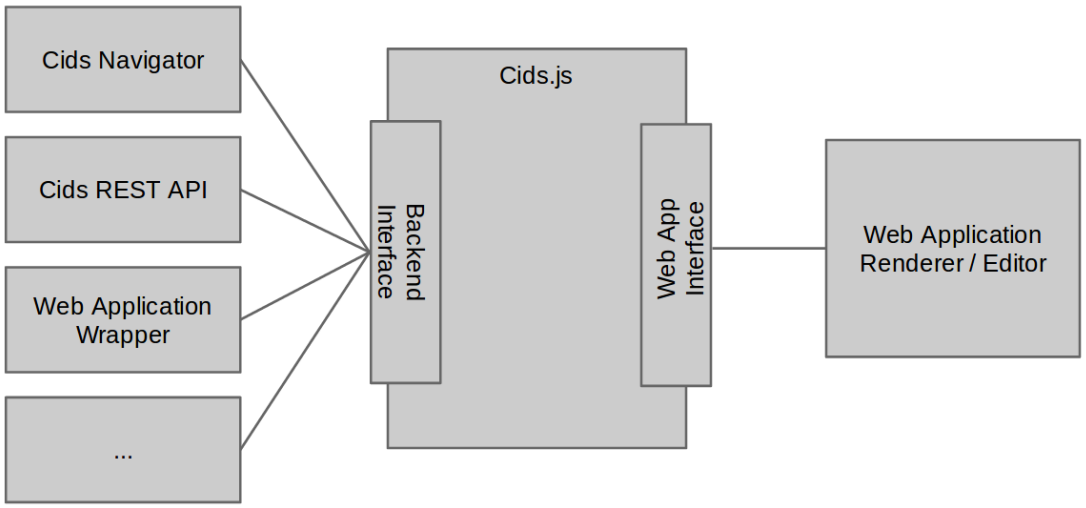


Figure 5.7: Architectural Overview CidsJS

necessary in fact the selected catalogue object also gets injected from the navigator into the editor. Just if this happens, it is possible to forward this data to the web application.

Since it is necessary to regard the usage of the web renderer component also without the navigator this behaviour needs to be abstracted from the application itself. This is reached by separating the web renderer component from the context it is executed in, with an additional layer. Figure 5.7 gives an overview of the architectural design.

The web renderer component shall only communicate with the cidsJS layer which abstracts the context the web renderer component is executed in. As depicted in figure 5.7 the cidsJS layer needs to define fix interfaces for both sides, the client side, and the backend side, hence especially the navigator needs to inject data into the application context and needs to register a bridge object for data exchange. The cidsJS layer is represented as a JavaScript object that can be easily accessed within the whole JavaScript context of the loaded page and thereby is easily accessible within the Java scope. A very common practice for this is to use an anonymous function that handles the initialisation of the cidsJS layer object and finally exposes this object to the global scope of the application afterwards. A similar approach is used in other frameworks such as JQuery or Prototype. Placing the initialisation function in a separate file it is sufficient to include this

file with a script tag in the index file of the application to use the cidsJS layer object. Owing to the exposion of the cidsJS object to the global JavaScript scope, it is very easy to get a reference of the cidsJS object in Java as listing 5.4 demonstrates.

Listing 5.4: Injecting the cidsBean to the JavaScript application

```
1 JSObject cidsJs = (JSObject)webEng.executeScript("ci");
2
3  cidsJs.call("injectCidsBean",
4                CidsBean.getCidsBeanObjectMapper().
                    writeValueAsString(bean));
```

The first thing that is implemented is the injection of the cidsBean which represents the selected catalogue object. For this cidsJS is extended with a respective method called `injectCidsBean`. When the setCidsBean method of the `HtmlWidgetEditor` is called, the `injectCidsBean` method can be used to provide the web application context with the necessary data. Sequence diagram 5.8 demonstrates the single steps that are executed to injected the `CidsBean` into the web application. The same approach can be used for setting the `isRenderer` property that indicates if the application shall be used in render or editor mode.

Injecting the cidsBean into the JavaScript context is only a part of the solution. The `SurveyPlanController` must also be adopted to the new situation. Therefore cidsJS is extended with a method `getCidsBean` that is called from the `SurveyPlanController` during the initialisation. Hence there is no guaranteed order that the`SurveyPlanController` is initialised after the `CidsBean` was injected into cidsJS, the initialisation of the controller needs to be delayed. Therefore the `getCidsBean` method returns a promise object. Promises are a very common concept in JavaScript to handle asynchronous events. A promise is an object that represents the result of an action that is performed asynchronously and may or may not finish at any given point in time. Angular already provides a promise implementation that allows to register so called callback-functions for different cases. Different callback-functions can be registered for the case the result is available or a failure has happened. The `$http` service, that was used to load relevant data from file so far, also uses Angular's promise implementation.

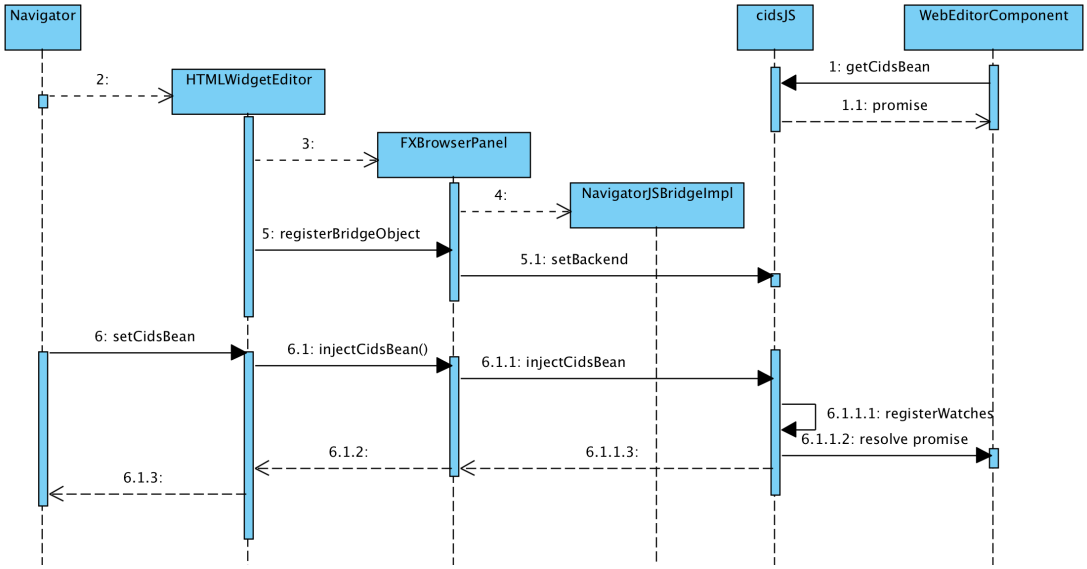


Figure 5.8: Injecting cidsBean into the web renderer component

Angular uses a so called defer object, that can be used to programmatically reject or resolve a promise which will execute the registered callback functions for the respective case. In our case this defer object is initialized during the initialisation of cidsJS and resolved when the injectCidsBean method is called. The sequence diagramm 5.8 demonstrates the steps that are executed during the initialisation process.

Figure 5.9 shows the usage of the developed web renderer component within the navigator. However, using the demo application as editor is still not possible in fact only a one way data exchange is implemented so far. Changing the data in the web editor component is out of the scope of the cids navigator. The following section describe how the data exchange from the web editor component back to the cids navigator is implemented.

The panel in that editors are embedded in the navigator offers buttons for saving and discarding changes that are made to the `CidsBean` object. If it was not changed, the editor is closed. In the case the object has changed, a dialog that calls on the user to confirm the action is shown. The `CidsBean` itself provides two different flags, that indicate if the `CidsBean` was changed. The first one, is set automatically if a value of a property has changed. The `CidsBean` therefore implements a `PropertyChangeListener` which allows to listen on every property

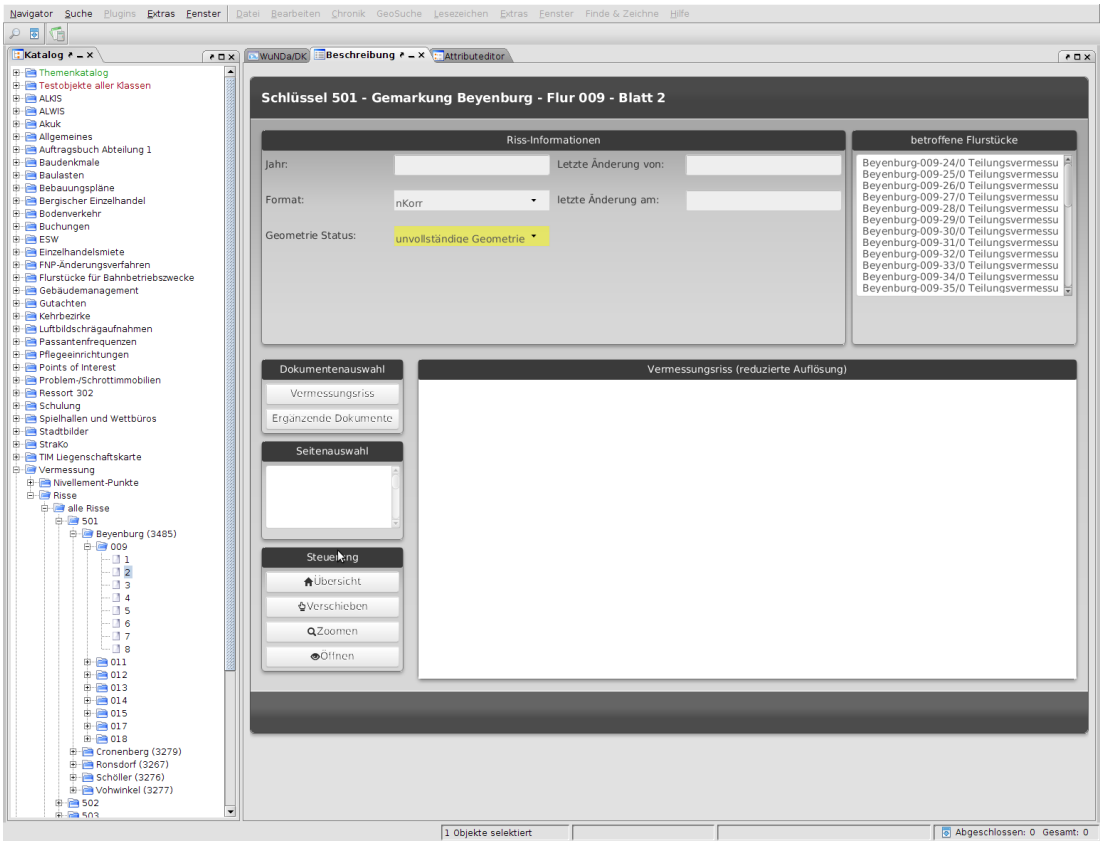


Figure 5.9: Navigator with Survey Plan Web Renderer

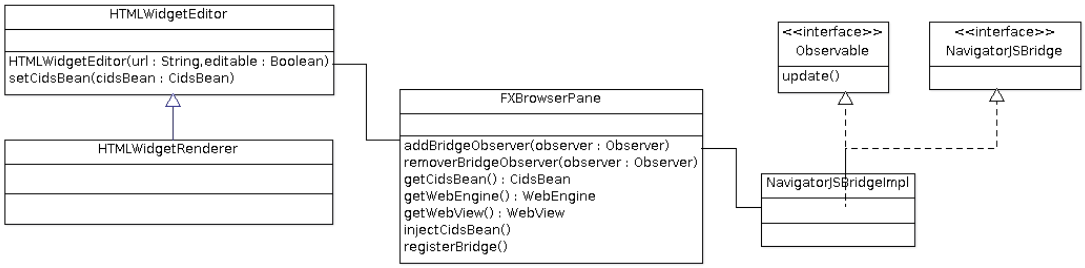


Figure 5.10: HTMLWidgetEditor and related classes

change. The second flag can be used to artificially mark the `CidsBean` as changed. The problem when using a web editor component, is that the save and discard actions are initiated by the user and within the context of the navigator. At this point the information if the `CidsBean` has changed needs to be already available. Therefore, simply setting the artificial change flag in the `HTMLWIdgetEditor` is not sufficient, in fact this marks the bean always as changed and modifies the behaviour of the save and discard button described above.

In contrast to the first approach the only possible way to ensure the same application logic, is to notify the navigator about every change of the `CidsBean` that happened in the web editor context. Hence this is only necessary when the application is used within the navigator, the cidsJS layer is extended with the respective functionality. When the `injectCidsBean` method in cidsJS is called, cidsJS registers Angular watches on every property of the cidsBean object. An Angular `$watch` are comparable with the `PropertyChangeListener` of the `cidsBean` and provide a simple listener mechanism for JavaScript objects. The `$watch` and `$watchCollection` method of an scope object can be used to register a callback function that is executed whenever the watch expression changes. The watch expression represents the value that will be watched which in our case this is the value of each property of the cidsBean. The callback function of a registered `$watch` then needs to notify the backend about that change. For this, the interface of the Java bridge object that allows up-calls from JavaScript to Java is extended with a respective method. To reflect the changes back into the cidsBean instance of the `HtmlWidgetEditor`, the bridge object also implements the Observable interface. Figure 5.10 and 5.11 visualises the single steps and classes that are involved.

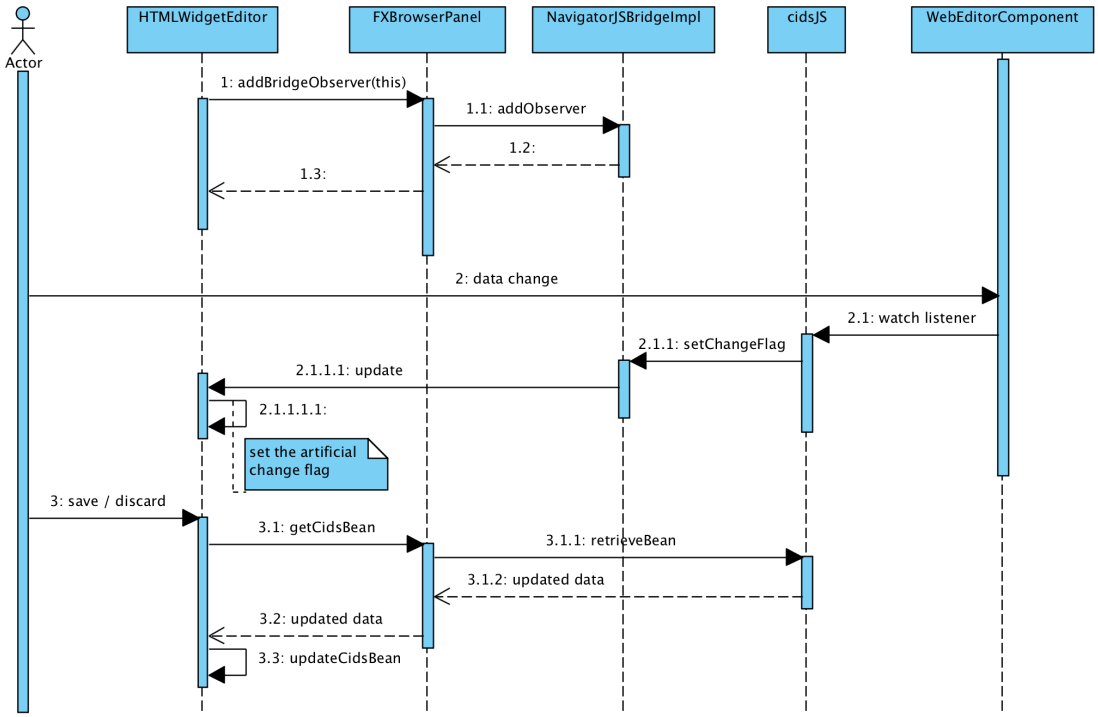


Figure 5.11: Retrieved CidsBean data from the web editor component

Testing the above described implementation shows that the numerous watches for every property on the `Cidsbean` slow down the performance of the web editor component. The reason for this lies in the architecture of Angular. The watch expressions are checked every time a property on the corresponding scope has changed. Hence the demo application mainly consists of one scope, all watch expressions are checked every time Angular checks the scope for changes. Portioning the application into multiple scopes would also solve the problem, but is in conflict with the implemented abstraction of the web application and the backend used. Hence the main reason of the watches isnjust to indicate that the data has changed, the watches are only needed until the first change in the data has happened. After that point the listener mechanism is no longer needed. Calling the `$watch` function returns a de-registration function that disables the listener mechanism for that watch. To solve the above mentioned problem, the de-register functions are saved in an array when the watches are set up. The callback function of the watches uses the bridge object to mark the `CidsBean` as changed. Afterwards, all de-register functions to disable watch-expressions on

are executed.

Hence the data changes are no longer submitted to the `HTMLWidgetEditor` the artificial change flag of the `CidsBean` is set when the data has changed. Furthermore it is necessary to retrieve the data from the web editor component in case the user wants to save the changes. The navigator already provides an interface `EditorSaveListener` that offers an editor the chance to execute some code before the data is persisted. By implementing this interface in the `HTMLWidgetEditor`, it is possible to retrieve the latest state of the `CidsBean` with the changes that were made. Hence the `NavigatorAttributeEditorGUI` uses its own reference of the `CidsBean` it is not possible to assign a new object to the `CidsBean` field of the `HtmlWidgetEditor` hence the reference of the `NavigatorAttributeEditorGUI` points still to the old `CidsBean` reference. Because it is not clear what properties of the `CidsBean` have changed, the `CidsBean` class is extended with a method `bulkUpdate(CidsBean otherBean)` that updates the properties of a `CidsBean` according to an other instance of a `CidsBean`.

Figure 5.12 depicts the usage of the demo application as editor with the final state of the implemented bi-directional data exchange.

5.4 Problems and Enhancements

5.4.1 Style of options list

With the implemented bi-directional data exchange first impressions using the developed survey plan demo application as renderer and editor can be gathered and show some minor problems that still can be improved.

The first issue is, that the option lists of select elements ignore custom css styles. Figure 5.13a depicts the options list of an select element in the JavaFX WebView. Although the implemenation of the options list is always browser dependent and therefore are only poorly customizable, the most modern browsers allow basic styling of option lists like setting the background color and so on. Unfortuneately the JavaFX WebView uses a JavaFX `ContextMenu` for implementing the options

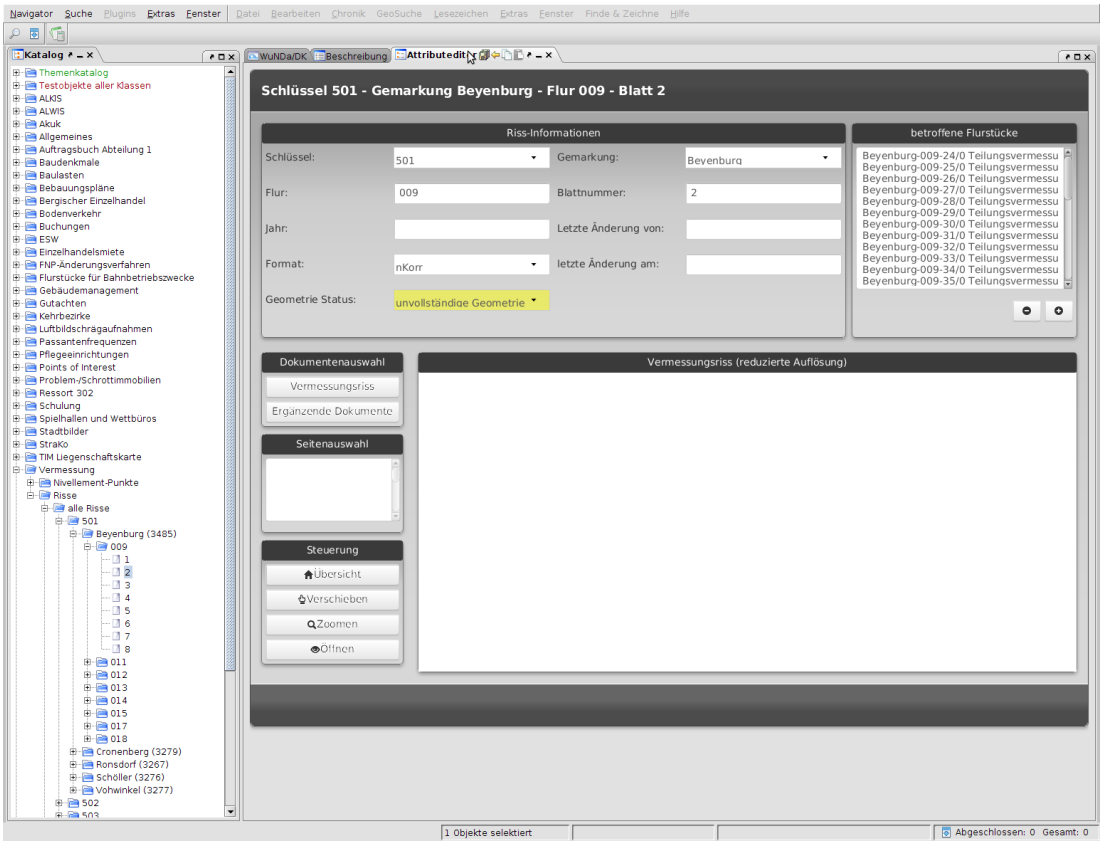
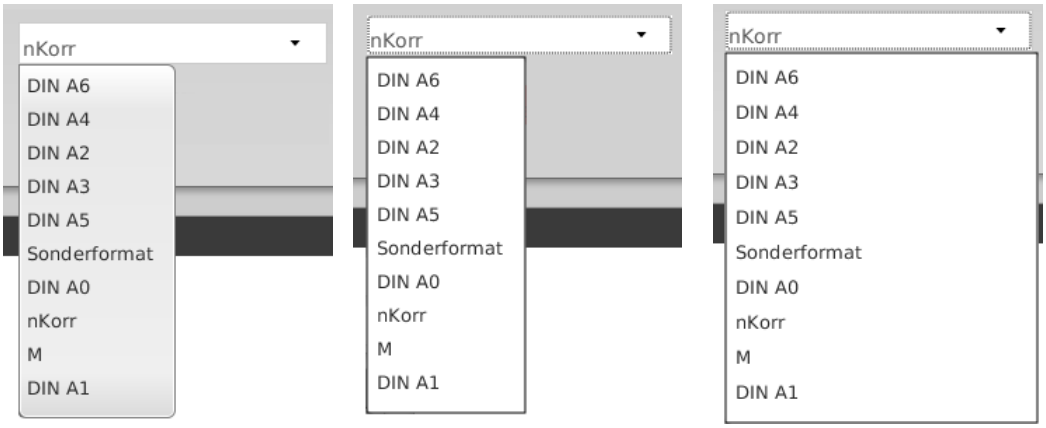


Figure 5.12: Navigator with Survey Plan Web Editor



(a) Normal State **(b) Custom CSS** **(c) Skin and CSS**

Figure 5.13: Design of options list for select elements

list of a select element and ignores CSS styles defined in the web application context.

One central approach of JavaFX is to use a custom implementation of cascading style sheets to separate the layout and content. This implementation is very similar to the CSS implementation used in HTML and relies on the W3C CSS standard 2.1. In [54] a good introduction on styling JavaFX applications is given. A possible solution for solving the above mentioned problem, is to use JavaFX style sheets to adopt the look of the `ContextMenu` class to better fit into the demo application. Following this approach does not solve the general problem but will provide a sufficient work around. As described in [54], the JavaFX style sheets are applied to a `Scene` object. To add styles to a `Scene` object it is sufficient to provide it with the path of the file that contains the style sheet definitions. The style itself needs to be defined for the class `.context-menu`. Figure 5.13b shows the options list with the custom CSS.

Although the look of the options list now fits more into the look of the demo application there is still the problem that the width of the options list is just as width as the largest element. A more appropriate visualisation is to extend the width to at least the width of the select element. The first idea to use the CSS width property pointed out, that the JavaFX style sheet implementation does not support the width property since layouting in JavaFX is implemented with layout manager objects similar to Swing. A possible workaround is to provide

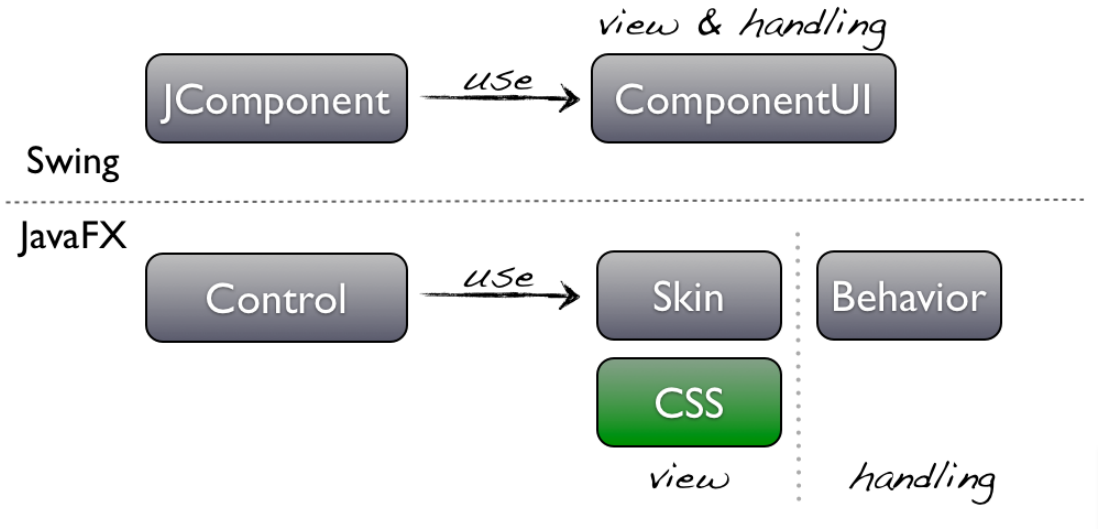


Figure 5.14: Architecture of javaFX Components in contrast to Java Swing, [55]

a custom skin class for context menus, that is used when a `ContextMenu` object is generated. A good explanation for this procedure can be found in [55]. To put it in a nutshell, every JavaFX component is composed by a control, a skin and a behaviour class. This approach is very similar to Java Swing where very component also has an UI class that combines the tasks of skin, control and behaviour. Figure 5.14 demonstrates this.

In the constructor of the custom skin class for context menus we wrap the existing menu items into an rectangle that has the same width of the select element and replace the menu items of the context menu. Figure 5.13c shows the final state of the options list for select elements.

5.4.2 Avoiding flickr effect during load

Another problem that can be regarded is that the web application is already visualised although it is not finally initialised. The user can observe a flickering effect of multiple different states, each one only present for a short time. Normally the description pane and the editor panel show a waiting symbol until the final renderer or swing component is initialized. In the case of web applications there are two more different states depicted in figure 5.15.

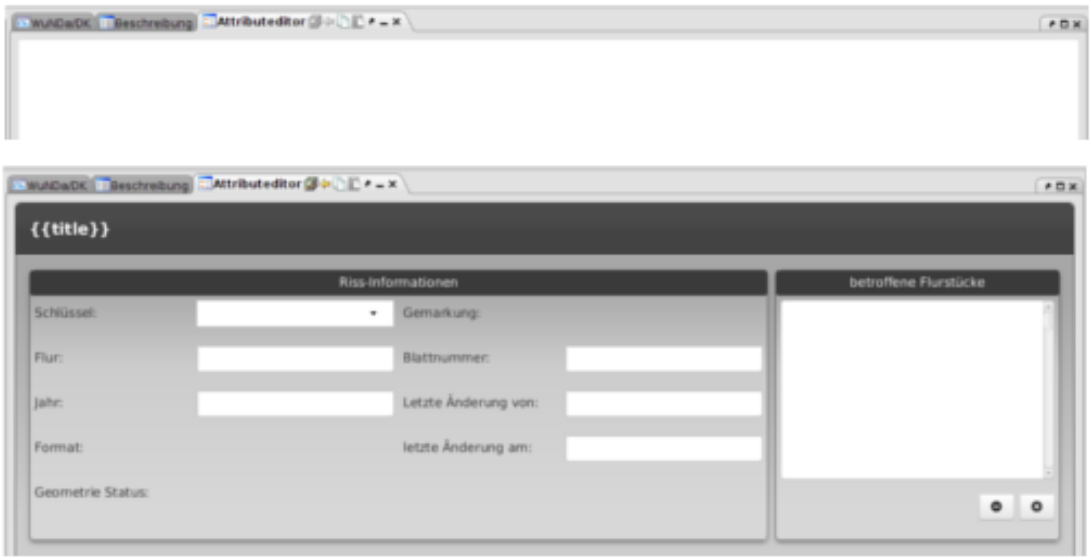


Figure 5.15: Flickering states

The first state is a totally white panel. This white panel represents the JavaFX WebView container. Hence the web page is not totally loaded at this point, the WebView just renders a simple white background. In the second state, the web page is completely loaded because the HTML and CSS parts of the application are already visible. But instead of the actual data, the user can see the binding expressions that are used in the HTML mark-up to enable the data binding in Angular. This is a very common problem in angular and is documented in [56], [57] and [58]. The problem is, that the templates are already visualised before they are compiled from Angular, or in other words, the web page is totally loaded, however the Angular application is not totally initialized. The Angular documentation offers two possibilities to work around this issue, the first one is to use a special directive `ng-cloak`. The Angular documentation states that the `ng-cloak` directive can be used to "[...] prevent the Angular html template from being briefly displayed by the browser in its raw (uncompiled) form while your application is loading. Use this directive to avoid the undesirable flicker effect caused by the html template display." [56] Unfortunately using this directive didn't work for a unknown reason. The second solution is to use the `ng-bind` directive instead of the double curly braces notation. Using the `ng-bind` directive hides the curly braces, however, the user still has a flickering effect when the

templates are compiled and the data is inserted.

To totally avoid those flickering effects another solution is needed. The main reason why those multiple states are visible to the user, is that the navigator assumes that the renderer respective the editor is completely initialized after the `setCidsBean` method is called. Despite this is true for Swing based renderer, this is not true for the `HtmlWidgetEditor`, hence the web page is asynchronously loaded. Thus, the renderer and editor loading mechanism needs to be blocked until the web application is completely initialized.

The first problem to solve when trying to implement the above outlined work around, is to determine when the angular application has finished the intitialisation of the application. In fact Angular does not fire a event for that case, it is necessary to understand how an angular application is bootstrapped. The chain of actions that happen during the initialisation are described in [59] but without going into great detail one of the latest things is setting up the controllers. In fact we have delayed the initialisation of the application to the point all relevant data is available the `SurveyPlanController` function is executed and after the Angular application is initialised. We can easily extend the `SurveyPlanController` to send a signal back to the navigator. In order to do so, the cidsJS layer and the interface of the bridge object are extended with corresponding methods.

Now that web editor component application notifies the bridge object as soon as it is initialised the loading process needs to be delayed up to this point. For this a new interface, named `InitialisationLocker`, is introduced that basically has a method that returns a `CountDownLatch` object. A `CountDownLatch` allows one or more threads to wait until a other thread completes its execution. It is initialized with a given value. Additionally the await method of the `CountDownLatch` will block until the value oof it reaches zero. The thread in that the `DescriptionPane` loads the renderer needs to check if the renderer that is currently loaded implements the `InitialisationLocker` interface and calls the await method of the `CountDownLatch` this interface provides. This causes to block the thread of the loading mechanism. If the web application sends the message that it is initialised, the `HtmlWidgetEditor` counts down the value of the `CountDownLatch` which reverses the blocked status renderer loading thread.

Testing the above mentioned changes shows an unexpected result. At the very beginning there is still a stutter with a white area until the web application is displayed. Further investigation on that issue shows that the JavaFX WebView starts its rendering process just in the case the WebView is visible. In [60] a possible solution is described. The idea is to do a snapshot of the WebView in the background. This initiates the internal rendering process.

6 Discussion

6.1 Conclusion

The navigator is successfully extended with the ability to integrate arbitrary web applications, which is one of the main objectives of the presented work. Withal, the degree how seamless an web application is integrated, is very scalable and reaches from the simple display of web applications to highly integrated components, that interact with the already existing Swing based navigator GUI and behave like already existing Swing renderer. This scalability ensures that the kind of web application that can be integrated is not restricted.

The implemented approach allows a very easy configuration to indicate when a web application shall be used as renderer and editor. Using the already existing class attributes it is possible to gain from the already existing support in the cids management tools. Hence the configuration is based on cids classes, it is possible to easily mix the usage of swing and web based renderer in one application.

Another beneficial side effect, is the introduction of the new JavaFX WebView as additional and optional browser API for description pages that offer additional information for a catalogue node. Hence the JavaFX is much more feature rich than the two browser API's that can be used up to now, totally new possibilities in the design of description pages are available now. An important key factor of the implementation is that the usage of the JavaFX WebView as browser for description pages is easily configurable with a new property in the navigators configuration file and is fully backward compatible. This is especially important because it can not be guaranteed that JavaFX is available on every client.

The developed web based survey plan demo application proves that a seamless integration of web applications can be reached. Important to note is that the developed application can be used in two different modes, an editor and a renderer mode. This bisects the development and maintenance efforts. With the implemented abstraction of the back-end from the application itself, it is possible to conduct the demo application also in other environments, for example a normal web server. Thereby, just a minimum amount of adoptions are necessary to get the application to work, which was another objective defined in chapter 1.2.

The implementation of the survey plan demo application and the integration into the cids navigator is preceded by an exhaustive examination of modern web development technologies with a special focus on JavaScript MVC frameworks, which are one of the latest trends in web development. The focus on these frameworks is explained by two different factors. The first one is, that those frameworks fit most conceptually into the cids architecture, hence all server side backend is already implemented or will be in the new RESTful server API. The second one and more important one is, that building feature rich and highly interactive web applications always require the usage of HTML, CSS and, first and foremost, JavaScript. Alternative technologies suffer from the easy and lightweight accessibility that web applications normally have since they are plugin based and can be regarded as deprecated technologies. In particular, the comparison of the MVC frameworks has revealed a very promising and powerful candidate, AngularJS. Again the development of the survey plan demo application has shown how easy web based renderer components can be implemented using AngularJS.

6.2 Future Work

Even if the reached state of the presented work is rather complete and quite useful, there are still some open issues and improvements that shall be solved in future work. The most obvious one is to implement the missing features in the survey plan renderer. To fully replace the existing Swing based renderer it is necessary to extend it with a zoom- an pan-able visualisation of the historic survey plan documents. Additionally, it must be possible to add land parcels to a survey plan. In the swing version, a dialog that supports the user choosing a land parcel exists. A similar functionality is also needed in the html version.

Another important issue that was not regarded in the implementation is to extend the survey plan application, to use different styles, depending on the context it is nested in. Within the navigator the design that emulates the Swing renderer is sensible. However, when the application is used standalone in a web browser or is used in an other context, a design more suitable to web pages is preferred. That this is possible in general shows the trend of responsive web design, which adopts the layout of the web application to the size of the device it is used. A possible approach to implement this, is to use browser sniffing which allows to react on the browser that is used.

Besides the open extensions of the demo application, there are as well some minor issues with the JavaFX WebView itself. The first one is that the visualisation of web applications in the WebView is slightly blurry and has not a good quality like other browsers. Unfortunately no solution or possible workaround can be found at present. There is an reasonable expectation that this problem is solved automatically with the next major release of the Java runtime environment version 8 since the official oracle roadmap points out enhancements of the WebView for that release. Possibly, the current workaround that styles the options list of select elements is no longer needed in that version and it is possible to define the style in the CSS context of the web application.

Regarding the bi-directional data exchange of the navigator and integrated web applications, it is clear that there is also open work to do. As outlined, the interface is based on the RESTful cids server API, but not fully implemented. There is also the chance to create a “cids-legacy implementation” of this API,

that only implements a subset of the cids REST API. This implementation can use already existing cids server functionality, but is accessed as RESTful API, which offers the chance to already develop complex web applications based on cids. The legacy implementation can be swapped with the final REST API later on, without changing the web applications.

From a more strategical point of view, the most important future work is to expedite the development of the new cids REST API. In strong relationship with that, it is important to develop first complex web application that make use of the new API. A possible and promising approach is to develop an application that emulates basic features of the already existing navigator, like the catalogue and rendering / editor mechanism. Such an application would be an ideal supplement of the cids platform and presented work.

List of Figures

Bibliography

[1] Cismet GmbH. *Cismet ReadMe*. 2013. URL: http://www.cismet.de/en/cidsReadme.html (visited on 11/20/2013).

[2] Wikipedia. *Comparison of web application frameworks*. 2014. URL: http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks.

[3] Jesse James Garrett. *Ajax: A New Approach to Web Applications*. 2005. URL: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/ (visited on 01/14/2014).

[4] Jeremy Allaire. *Macromedia Flash MX—A next-generation rich client*. 2002. URL: http://download.macromedia.com/pub/flash/whitepapers/richclient.pdf.

[5] Alessandro Bozzon et al. "Capturing RIA concepts in a web modeling language". In: *WWW '06: Proceedings of the 15th international conference on World Wide Web*. Edinburgh, Scotland, 2006, pp. 907–908. DOI: http://doi.acm.org/10.1145/1135777.1135938.

[6] Marianne Busch and Nora Koch. *Rich Internet Applications State-of-the-Art*. Tech. rep. Ludwig-Maximilians-Universität München, Germany, 2009. URL: http://uwe.pst.ifi.lmu.de/publications/maewa_rias_report.pdf.

[7] Alessandro Bozzon et al. "T.: Conceptual Modeling and Code Generation for Rich Internet Applications". In: *ICWE2006: International Conference on Web Engineering*. ACM Press.

[8] Tom Noda and Shawn Helwig. "Rich internet applications". In: *Technical Comparison and Case Studies of AJAX, Flash, and Java based RIA UW E-Business-Consortium Opinion Papers* (2005).

[9] Alexander Hofmann. "Silverlight vs. Flash - zwei Namen für eine Technologie". In: (2011). URL: http://www.uni-bamberg.de/fileadmin/uni/fakultaeten/wiai_lehrstuehle/medieninformatik/Dateien/Sammelbaende/MI-Sem_So2011_Web-Technologien.pdf#page=17.

[10] Yakov fain. *Rich Internet Applications: State of the Union*. 2007. URL: http://flexblog.faratasystems.com/2007/02/13/rich-internet-applications-state-of-the-union (visited on 02/08/2014).

[11] Steve Jobs. *Thoughts on Flash*. 2010. URL: http://www.apple.com/hotnews/thoughts-on-flash/ (visited on 01/08/2014).

[12] w3techs. *Usage of client-side programming languages for websites*. 2010. URL: http://w3techs.com/technologies/history_overview/client_side_language/all/y (visited on 01/08/2014).

[13] Nicholas C. Zakas. *How many users have JavaScript disabled?* 2013. URL: http://developer.yahoo.com/blogs/ydn/many-users-javascript-disabled-14121.html (visited on 01/14/2014).

[14] Glenn E. Krasner and Stephen T. Pope. "A Cookbook for Using the Model-view Controller User Interface Paradigm in Smalltalk-80". In: *J. Object Oriented Program.* 1.3 (Aug. 1988), pp. 26–49. ISSN: 0896-8438. URL: http://dl.acm.org/citation.cfm?id=50757.50759.

[15] Golo Roden. *Model View ViewModel mit Knockout.js*. 2013. URL: http://www.heise.de/developer/artikel/Model-View-ViewModel-mit-Knockout-js-1928690.html (visited on 01/14/2014).

[16] Igor Minar. *MVC vs MVVM vs MVP*. 2012. URL: https://plus.google.com/+AngularJS/posts/aZNVhj355G2 (visited on 01/14/2014).

[17] Marius Gundersen. *A comparison of the two-way binding in AngulaJS, EmberJS and KockoutJS*. 2013. URL: http://2013.jsconf.eu/speakers/marius-gundersen-a-comparison-of-the-twoway-binding-in-angularjs-emberjs-and-knockoutjs.html (visited on 01/14/2014).

[18] Google Inc. *Angular Developer Guide - Data Binding*. 2013. URL: http://docs.angularjs.org/guide/databinding (visited on 02/12/2014).

[19] Jose Maria Arranz Santamaria. *The Single Page Interface Manifesto*. 2011. URL: http://itsnat.sourceforge.net/php/spim/spi_manifesto_en.php (visited on 01/14/2014).

[20] *Can I use...* 2013. URL: http://caniuse.com/#search=history (visited on 01/14/2014).

[21] *Breaking the Web with hash-bangs*. 2011. URL: http://isolani.co.uk/blog/javascript/BreakingTheWebWithHashBangs/ (visited on 01/14/2014).

[22] *It's About The Hashbangs*. 2011. URL: http://danwebb.net/2011/5/28/it-is-about-the-hashbangs (visited on 01/14/2014).

[23] *Thoughts on the Hashbang*. 2011. URL: http://www.adequatelygood.com/Thoughts-on-the-Hashbang.html (visited on 01/14/2014).

[24] W3C. *HTML5 - A vocabulary and associated APIs for HTML and XHTML*. 2013. URL: http://www.w3.org/TR/html5/browsers.html#history-1 (visited on 01/14/2014).

[25] Google Inc. *Angular Developer Guide - Using $location*. 2012. URL: http://docs.angularjs.org/guide/dev_guide.services.\$location (visited on 01/15/2014).

[26] Google Inc. *Making AJAX Applications Crawlable - Getting Started*. 2012. URL: https://developers.google.com/webmasters/ajax-crawling/docs/getting-started (visited on 01/15/2014).

[27] J. Gettys R. Fielding UC Irvine et al. *RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1*. 1999. URL: http://www.faqs.org/rfcs/rfc2616.html (visited on 02/04/2014).

[28] Sander Versluys. *What is the maximum length of a URL in different browsers?* 2013. URL: http://stackoverflow.com/questions/417142/what-is-the-maximum-length-of-a-url-in-different-browsers/7056886#7056886 (visited on 02/04/2014).

[29] Oracle. *Java Scripting Programmer's Guide*. 2011. URL: http://docs.oracle.com/javase/6/docs/technotes/guides/scripting/programmer_guide/#jstojava (visited on 02/04/2014).

[30] *Jetty - Quick Start Guide*. 2014. URL: http://www.eclipse.org/jetty/documentation/current/quick-start.html (visited on 02/04/2014).

[31] Google Inc. *Google Web Toolkit*. 2014. URL: http://www.gwtproject.org/ (visited on 02/04/2014).

[32] *Flying Saucer*. URL: https://code.google.com/p/flying-saucer/ (visited on 02/06/2014).

[33] *Lobo Browser API*. URL: http://lobobrowser.org/browser/api-info.jsp (visited on 02/06/2014).

[34] Lars Gunther. *Acid3 receptions and misconceptions and do we have a winner?* 2008. URL: http://www.webstandards.org/action/acid3/ (visited on 02/06/2014).

[35] Niels Leenheer. *HTML5 Test - how well does your browser support html5?* 2013. URL: http://html5test.com/ (visited on 02/06/2014).

[36] *THE CSS3 TEST*. 2013. URL: http://html5test.com/ (visited on 02/06/2014).

[37] Wikipedia. *ECMAScript*. 2014. URL: http://en.wikipedia.org/wiki/ECMAScript (visited on 02/06/2014).

[38] *ECMAScript Language test262*. 2010. URL: http://test262.ecmascript.org/ (visited on 02/06/2014).

[39] Brian Moschel. *CanJS: The Best of Both Worlds*. 2012. URL: http://de.slideshare.net/moschel/canjs-the-best-of-both-worlds (visited on 12/06/2013).

[40] soundstep. *What is soma.js?* URL: http://somajs.github.io/somajs/site/ (visited on 12/06/2013).

[41] Golo Roden. "Rückgrat geben - Model View Controller mit Backbone.js". In: *Heise Developer* (2013).

[42] Yehuda Katz. *Amber.js (formerly SproutCore 2.0) is now Ember.js*. 2011. URL: http://yehudakatz.com/2011/12/12/amber-js-formerly-sproutcore-2-0-is-now-ember-js/ (visited on 01/14/2014).

[43] TOM DALE YEHUDA KATZ. *STABILIZING EMBER DATA*. 2013. URL: http://emberjs.com/blog/2013/03/22/stabilizing-ember-data.html (visited on 01/14/2014).

[44] Tilde Inc. *Ember Guides - Defining you routes*. 2013. URL: http://emberjs.com/guides/routing/defining-your-routes/ (visited on 01/14/2014).

[45] Tilde Inc. *Ember Guides - Controllers*. 2013. URL: http://emberjs.com/guides/controllers/ (visited on 01/14/2014).

[46] Tilde Inc. *Ember Guides - The View Layer*. 2013. URL: http://emberjs.com/guides/understanding-ember/the-view-layer/ (visited on 01/14/2014).

[47] Google. *Angular Developer Guide - Introduction*. 2013. URL: http://docs.angularjs.org/guide/introduction (visited on 01/14/2014).

[48] Google. *Angular Developer Guide - What is a Module?* 2013. URL: http://docs.angularjs.org/guide/module (visited on 01/14/2014).

[49] Google. *Angular Developer Guide - What are Scopes?* 2013. URL: http://docs.angularjs.org/guide/scope (visited on 01/14/2014).

[50] Google Inc. *Angular Developer Guide - Conceptual Overview*. 2013. URL: http://docs.angularjs.org/guide/concept (visited on 02/12/2014).

[51] Misko Havery. *Databinding in angularjs*. 2013. URL: http://stackoverflow.com/questions/9682092/databinding-in-angularjs (visited on 01/14/2014).

[52] Randahl Fink Isaksen. *Why isn't JavaFX on the jdk1.7.0$_u$10build12classpath?*. 2012. URL: http://mail.openjdk.java.net/pipermail/openjfx-dev/2012-October/004045.html (visited on 02/11/2014).

[53] *LiveConnect Support in the New Java^{TM} Plug-In Technology*. 2013. URL: https://jdk6.java.net/plugin2/liveconnect/#JS_JAVA_CONVERSIONS (visited on 02/11/2014).

[54] Alexander Kouznetsov Joni Gordan. *Skinning JavaFX Applications with CSS*. 2013. URL: http://docs.oracle.com/javafx/2/css_tutorial/jfxpub-css_tutorial.htm (visited on 02/03/2014).

[55] Hendrik. *Custom UI Controls with JavaFX – Part 1*. 2012. URL: http://www.guigarage.com/2012/11/custom-ui-controls-with-javafx-part-1/ (visited on 02/03/2014).

[56] *ngCloak - directive in module ng*. 2013. URL: http://docs.angularjs.org/api/ng.directive:ngCloak (visited on 02/03/2014).

[57] *ngBind - directive in module ng*. 2013. URL: `http://docs.angularjs.org/api/ng.directive:ngBind` (visited on 02/03/2014).

[58] *Prevent double curly brace notation from displaying momentarily before angular.js compiles/interpolates document*. 2013. URL: `http://stackoverflow.com/questions/12866447/prevent-double-curly-brace-notation-from-displaying-momentarily-before-angular-j` (visited on 02/03/2014).

[59] Google Inc. *Angular Developer Guide - Bootstrap*. 2013. URL: `http://docs.angularjs.org/guide/bootstrap` (visited on 02/11/2014).

[60] *Strange behaviour on preloading web page in WebView*. 2013. URL: `https://community.oracle.com/thread/2584040` (visited on 02/11/2014).