jamk.fi

Infrastructure and asset management using modern web-technologies

Henri Kovanen

Bachelor's thesis
December 2014

Degree Programme in Media Engineering
School of Technology, Communication and Transport

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Description

Author(s) Kovanen, Henri	Type of publication Bachelor's thesis	Date 10.12.2014
	Number of pages 40	Language of publication: English
		Permission for web publication: x

Title of publication
Infrastructure and asset management using modern web-technologies

Degree programme
Media Engineering

Tutor(s)
Manninen, Pasi

Assigned by
Versine Ltd.
Aho, Kari

Abstract

The main goal of the thesis was a complete technological overhaul of an existing web-application focusing on the front-end of the application, assigned by Versine Ltd. The thesis examines different client-side front-end frameworks used in single page web-applications and the aim was to choose the most appropriate for the assignment.

Several user interface and application architecture frameworks were examined and compared, and the most suitable ones were selected. Additional supportive frameworks and libraries were also chosen. As a result a full front-end development stack for creating user interfaces, developing the business logic and automating the release process was created.

This thesis presents an example where the selected set of tools is used for creating a feature for controlling listings of prices between companies. The example will follow the development process used at Versine Ltd and it describes all the steps from the initial planning of the feature to releasing and distributing of a new version of the application with the implemented feature. The implemented feature has been put into production and it has been actively used since.

Possible future development on the results was contemplated on and it is described in the thesis. The price list will be somewhat tweaked; however, the development stack will still need some additional features, for example unit testing. The conclusion contains reflection on how the assignment succeeded and what was learned from the author's perspective.

Keywords/tags (subjects)

HTML, JavaScript, Single page application

Miscellaneous

Kuvailulehti

Tekijä(t) Kovanen, Henri	Julkaisun laji Opinnäytetyö	Päivämäärä 10.12.2014
	Sivumäärä 40	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Infrastructure and asset management using modern web-technologies

Koulutusohjelma
Mediatekniikka

Työn ohjaaja(t)
Manninen, Pasi

Toimeksiantaja(t)
Aho, Kari
Versine Oy

Tiivistelmä

Opinnäytetyön tavoitteena oli Versine Oy:n olemassa olevan www-sovelluksen front-end teknologioiden uudistaminen. Työ tutkii erilasia front-end kirjastoja, joita käytetään yleisesti yhden sivun applikaatioiden kehityksessä. Tavoitteena oli valita toimeksiantoon sopivat kirjastot ja kehittää niiden avulla järjestelmään uusi ominaisuus.

Työssä tutustuttiin useaan käyttöliittymä- ja sovellusarkkitehtuurikirjastoon ja niitä vertailemalla valittiin toimeksiantoon sopivat työkalut. Toimeksiantajan toiveena oli myös, että työkalukokoelmaa voisi käyttää yrityksen tulevissa projekteissa, joten joustavuus ja modulaarisuus oli tärkeässä asemassa. Edellä mainittujen kirjastojen lisäksi niitä tukemaan valittiin kirjastoja esimerkiksi riippuvuuksien hallintaan sekä DOM-manipulaatioon. Valintojen tuloksena on front-end työkalukokoelma, jota voi käyttää käyttöliittymien- ja toimintalogiikan kehitykseen sekä julkaisun automatisoimiseen.

Esimerkkinä työssä kehitettiin valituilla työkaluilla Versine Oy:n olemassa olevan kehitysprosessin mukaisesti hinnastomoduli. Prosessin kulku käydään läpi suunnittelusta julkaisuun. Hinnastomoduli integroitiin olemassa olevaan järjestelmään. Hinnastomodulin avulla hinnastoja voi luoda ja lähettää yritysten välillä. Kehitetty moduli on ollut valmistumisestaan asti aktiivisessa tuotantokäytössä.

Työssä käsitellään myös tulosten jatkokehitystä. Hinnastomodulia tullaan muokkaamaan asiakkailta saadun palautteen mukaisesti, jos tarve niin vaatii. Tällä hetkellä työkalukokoelma ei kuitenkaan ole kaikenkattava ja vaatii lisäominaisuuksia esimerkiksi yksikkötestausta varten. Pohdintaosassa käydään läpi kuinka toimeksianto sujui ja mitä kirjoittaja oppi työn kirjoittamisen aikana.

Avainsanat (asiasanat)
HTML, JavaScript, Single page application

Muut tiedot

CONTENTS

TERMS AND ABBREVATIONS 3
1 INTRODUCTION 6
1.1 Objectives 6
1.2 Client 6
1.3 What is MDO? 7
2 SINGLE PAGE APPLICATIONS 8
2.1 Single page applications on the web 8
2.2 A look into MVC frameworks 9
2.3 Server 11
3 CHOOSING THE FRAMEWORKS 12
3.1 Selecting the MVC framework 12
3.1.1 Basis on the selection 12
3.1.2 Maturity 13
3.1.3 Community 14
3.1.4 General suitability 15
3.1.5 Summary of considered MVC frameworks 16
3.2 Selecting the UI framework 16
3.3 Selecting other needed libraries and tools 19
4 FEATURE DEVELOPMENT PROCESS 20
4.1 Feature-driven development 20
4.2 Example case: price lists 21
4.3 UI-design and implementation 22
4.4 Front-end implementation 26
4.4.1 Model implementation 26
4.4.2 View implementation 27
4.5 Back-end specification 31
4.6 Feature inspection 31
4.7 Deployment 33
5 CONCLUSION 35
5.1 Results 35
5.2 Analysis and project continuation 37
REFERENCES 39

FIGURES

Figure 1. Growth of transferred JavaScript file size and request amounts. 9
Figure 2. MVC flowchart. 10
Figure 3. Structure of the price list feature 21
Figure 4. Inconsistency marked in red. It should be in the group marked in blue. 23
Figure 5. Create button added to the button group to remove inconsistency 23
Figure 6. The form was moved into a modal to remove the inconsistency. 24
Figure 7. The company price list view 28
Figure 8. Add row modal with invalid attribute 30
Figure 9. Bitbucket's pull request interface 32
Figure 10. Example of a code comment in Bitbucket 33
Figure 11. A failed grunt process 34

TABLES

Table 1. Scores for framework maturity 14
Table 2. Scores for framework community 15
Table 3. Scores for framework general suitability 16
Table 4. Summary of MVC framework comparison 16

TERMS AND ABBREVATIONS

AJAX

Asynchronous JavaScript and XML. AJAX is a set of techniques to create asynchronous (in the background) HTTP requests. AJAX requests do not interfere with the UI or other programming logic. Despite its name, XML is not always used and JSON is often used instead of XML.

API

Application programming interface. When used in the web context an API is a definition of the structure of messages used to communicate between two systems. Usually the front-end and the back-end of an application.

BACK-END

Back-end refers to the server side functionality.

BOILERPLATE CODE

Boilerplate code is repetitive code which needs to be used in many places without alteration.

BITBUCKET

Bitbucket is a version control system based hosting site. Projects using Git or Mercurial can be hosted on Bitbucket. In addition to version control Bitbucket provides teams with collaboration tools like pull requests and an issue tracker.

CSS

Cascading Style Sheets. CSS is a style sheet language used for describing the look of a document written in a mark-up language. Most commonly used with HTML, but can be used with any kind of XML document.

CRUD

Create, read, update and delete are the four basic functions of persistent storage.

DOM

Document Object Model is a convention for representing and interacting with XML based documents. A single element is called a node and the nodes are organized in to

a tree structure called the *DOM tree*.

FRONT-END

Front-end refers to the functionality in the client side e.g. browser.

GIT

Git is a distributed version control system.

GITHUB

GitHub is a version control based hosting site similar to Bitbucket.

GZIPPED

GZIP is an application used for compressing and decompressing files. GZIP is widely used in UNIX systems in general, but in a web context it is used to compress HTTP requests and responses.

HTML

HyperText Markup Language is the most used markup language used to build web pages.

JSON

JavaScript Object Notation is a standardized format of data represented by key (or attribute or name)-value pairs. Commonly used in transferring data between a served and a web application.

MODAL

A modal is a graphical sub-element which disables the main element's use for long as it is active. Frequently used in displaying images in full size or when something important needs user interaction.

PULL REQUEST

Pull request is a workflow method used in conjunction with a version control system where a developer submits a change for evaluation. Pull requests enable discussion and review before the submitted change will be merged or rejected from the main repository.

REST

Representational state transfer in the web application context is an architectural style that can be used to build a web API. REST is most commonly used with HTTP and resources are identified with a unique URI (or URL). If HTTP is used, the resources are manipulated with HTTP standard methods (GET, PUT, POST and DELETE).

SOAP

SOAP is a protocol used to transfer structured data in web services. SOAP provides a uniform way for systems to communicate over HTTP using XML.

STACK OVERFLOW

Stack Overflow is a web site for programmers to ask and answer questions. In Stack Overflow anyone can ask a question and anyone can answer. The community then decides which the best answer is by voting.

UI

User Interface is the visual representation of a program. UI elements represent actions or commands, which are invoked after the user interacts with the element.

XSS-ATTACK

Cross-site scripting is a security vulnerability that can be found in web applications. XSS enables an attacker to inject a client side script into a web page.

1 INTRODUCTION

1.1 Objectives

The objective of this thesis was to research different front-end frameworks for UI-design and application architecture to be used in a web application. As a result of the research a set of front-end technologies were selected to provide a full stack of tools for development and deployment. The theoretical part focuses on examining different frameworks and tools and selecting the most suitable ones, while the hands on part will describe the development process and as an example a price list feature is implemented with the selected tools. As a result of the thesis a set tools were selected and combined to produce a stack for front-end development. In addition a production ready feature is produced. The MDO suite consists of four different applications described later; the thesis is focused on the MDO Hub application.

1.2 Client

MDO Group Ltd. is the new name of a mobile and server technology company, which continues the business operations of Telonet Ltd. and Versine Ltd. MDO Group offers software targeted at telecommunication and electrical engineering, energy infrastructure building, monitoring and maintenance for operators and device manufacturers and suppliers. MDO Group Ltd.'s head office is in Jyväskylä and other offices are situated in five different locations around Finland (Aho 2014).

Versine Ltd, the majority shareholder of MDO Group focuses on providing consumers and businesses with modern cloud and mobile solutions. The company has developed consumer applications that have been downloaded worldwide more than 60 000 times and the applications has been discussed in radio programs and have been recommended by national operators. Versine has also developed enterprise solutions for the needs of operators, device manufacturers, contractors and logistics sector (Aho 2014).

Magister Solutions Ltd, the majority shareholder of Versine, offers internationally recognized services ranging from developing new concepts and algorithms in device

development to software solutions satisfying the high requirements of space technology. The privately owned company was founded in 2005 with the intention to conduct research and development of wireless data radio interfaces, radio resources, radio simulations and networks. Magister Solutions currently employs about 30 leading ICT industry professionals and together with Versine and MDO Group employs about 40 people. The multinational and highly trained staff of Magister contains over 60% professors, doctors and doctoral students who in their daily work focus on solving the challenges of world's leading mobile, software and space technology suppliers and operators. Since its foundation, Magister Solutions has provided Nokia, Nokia Siemens Networks (NSN), and as of fall 2012 European Space Association (ESA) and Renesas Mobile Corporation with long supply contracts in expert and research services of future wireless network and devices. The responsibilities of the subsidiaries of Magister, Versine and MDO Group, are to develop and market modern mobile and cloud server solutions to the use of operators and device manufacturers (Aho 2014).

1.3 What is MDO?

MDO is a suite of modular online and mobile solutions for infrastructure lifecycle management from site planning to monitoring and maintenance. The MDO suite consists of four applications that complement each other in covering the whole process:

1. **MDO Mobile** – Is an Android application for on-site usage, it contains user specific work flows / packages with documentation tools. Users can have new tasks assigned for them in real time, with detailed instructions on what to do. MDO mobile is Interoperable with MDO Hub.

2. **MDO Hub** – Is an online service with workflow, asset and project management. MDO Mobile-data is collected and processed in real time for up to date information with comprehensive documentation tools.

3. **MDO Analyzer** – Is an application for wireless network testing built on Android. It provides mobile network, Wi-Fi and performance measurements with location information. The measurements are visualized live and

uploaded to MDO Visualizer for a more detailed analysis.

4. **MDO Visualizer** – Is an online service for performance analysis based on data collected by MDO Analyzer. There are modules for highlighting problem areas on a map or by a network base station.

As this thesis focuses on the MDO Hub application, other parts of the suite will not be covered within the scope of this thesis.

MDO hub is a cloud service for infrastructure and work management. The data within the service can be customized straight from the UI just as the client needs – without limiting the content or the amount of data. The application is used with an internet browser and can be accessed from anywhere at any time and users can send work assignments straight to a worker using the mobile application. The system also has modules for projects, price lists, orders and billing for large-scale management.

2 SINGLE PAGE APPLICATIONS

2.1 Single page applications on the web

"An SPA (Single page application) is an application delivered to the browser that does not reload the page during use" (Mikowski & Powell 2013, 4). The aim is to serve the user an experience that is as close to a native application as possible. Traditional web-applications navigate from a page to another after when the user interacts with the application. Form submissions and page navigations trigger page reloads and the user has to wait. If the page is heavy and the loading times are too long, users get frustrated and lose their train of thought. A delay between 0.2 and 1.0 seconds is a noticeable delay, and the user feels that the computer is "working" on something (Nielsen 1993). After the 1.0 second mark, users get the feeling that the interface is slow or sluggish. After user interaction SPAs manipulate a fragment of a page, so the amount of transferred may be significantly smaller or nonexistent. SPAs can be built to be fully loaded in the initial page load or to load fragments or whole modules

when they are needed. As browsers are constantly evolving to be more powerful, the use of JavaScript to build web-applications is growing. Between September 15. 2011 and 15. September 2014 deployed JavaScript code has grown from 148kB to 296kB (HTTP Archive, 2014), a growth of 100% (see Figure 1).

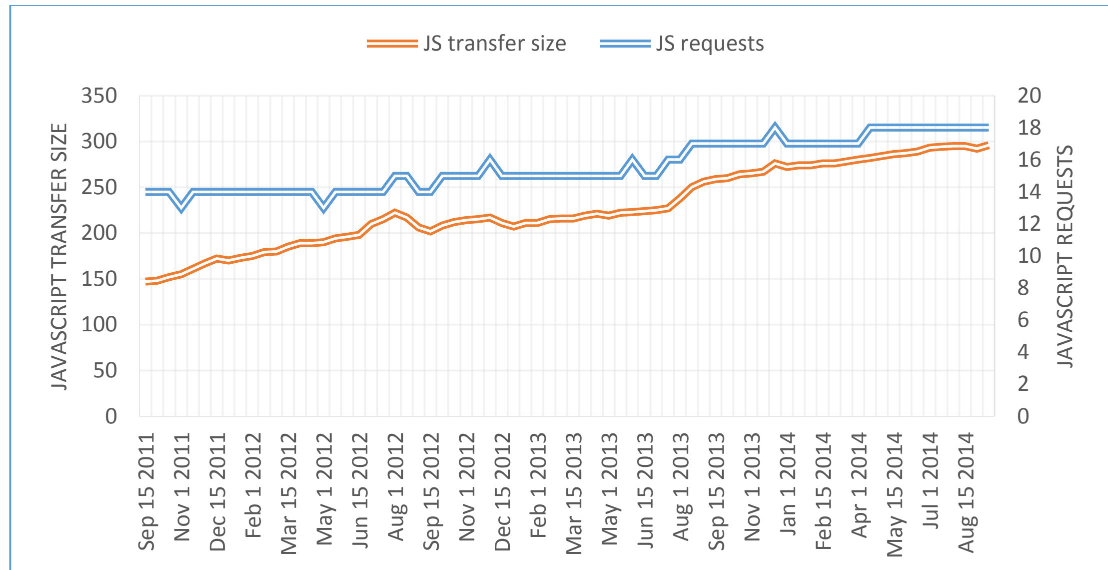

Figure 1. Growth of transferred JavaScript file size and request amounts.

As the complexity and size of an application grows, the need for a design pattern to keep the code maintainable and reusable is crucial. Using design patterns and developing applications to be modular is the key in developing a user friendly and native-like web application. It is logical to use a pattern originally designed for native applications to solve problems that occur in SPAs. One of the first and most used architectural patterns for implementing user interfaces is MVC. MVC was one of the first and most influential architectural design patterns for user interface design.

2.2 A look into MVC frameworks

MVC was first described in 1979 and in 1988 MVC was released as a general concept in The Journal of Object Technology. JavaScript has developed in to numerous frameworks that have support for MVC (or variations, which are referred to as the MV* family) (Osmani 2012). MV* patterns share some fundamental concepts with the classical MVC pattern, about how the logic in applications should be separated (see Figure 2).

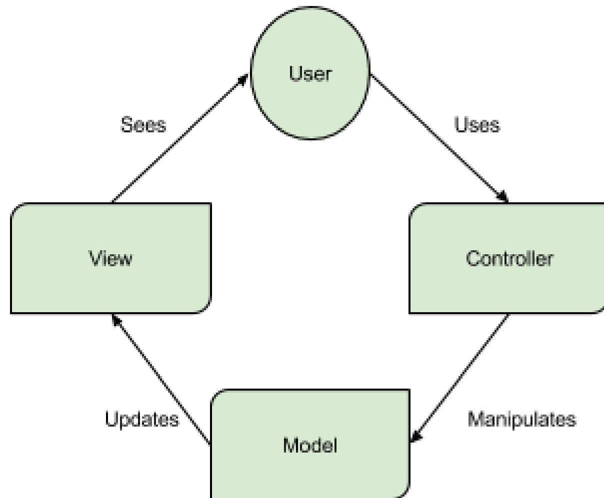

Figure 2. MVC flowchart.

Originally used in desktop applications coded in Smalltalk-80, MVC quickly increased its popularity and has become very popular in web-applications. There are many commercial and open-source frameworks to choose from for the front- and backend of a web application. MVC is widely used in large and complex web-applications, thus it was natural that MVC was chosen as the architecture. Some frameworks are pure MVC frameworks with all of the three components; however, as needs and technologies develop MVC has evolved into many different architectures like MVP, MVVM and MV*. Differences between them are not substantial, the basic idea of separating the data from the presentation is the same.

MVC splits the application structure in to three components: model, view and controller. By separating the structure in three components, developers can reduce the complexity of their programs and reuse as much logic as possible. Along with separating the business logic from the interface, MVC specifies the communication between components. Models contain the data used in an application. Views are interface elements in which the models are presented. Controllers manipulate models and views - user interactions are handled and stored in the associated models or model changes are sent to its associated view to represent changes that were made.

Developers can benefit from MVC in many ways; the code is divided into separate modules so that modifications are simpler. If data changes, the model needs to be

changed, or if a UI needs to be modified only the template needs to change. Developers can work on the UI and business logic separately, thus, multiple members of the team can work on a feature simultaneously.

2.3 Server

Traditionally web applications have been server based, where static HTML pages were sent to the user and JavaScript was only used to increase usability. In SPAs the page can be rendered with JavaScript. “The most common responsibilities of an SPA web server include authentication and authorization, data validation, and data storage and synchronization” (Mikowski & Powell 2013, 230). Decisions on how much data processing will be shared between the front- and backend should be made on a case-by-case basis. Strictly looking at server maintenance cost it would be ideal to process all data in the front-end; if heavy tasks can be processed on the user’s devices the server only needs to validate and store the data.

Doing all processing in the front end is a great and a doable idea, if the application is light. If the data set used by the application is complicated or large there will be problems - the processing power of the user’s device can vary a lot. If the target group of the application is large, the developer needs to take different kinds of users into consideration. There are many different web browsers, operating systems and devices, the different combinations are endless. To ensure the usability of the application for as many people as possible it is sensible to leave the heavy lifting for the server.

Operating system or programming language in the server does not matter, because usually data is transferred using AJAX to transfer JSON or XML data. The type of web service does not matter either, it can be a: REST, SOAP, or almost any other type of service.

3 CHOOSING THE FRAMEWORKS

3.1 Selecting the MVC framework

3.1.1 Basis on the selection

There are plenty of open source front-end frameworks to choose from: some feature a full development stack and others provide the bare essentials. Finding a good combination is essential for the project to succeed. Migrating from an MVC library to another may not sound like a substantial task, but staying on schedule is important as there are existing and new customers waiting for a new release of a product.

In the case of MDO the backend and mobile-app are already in production the frameworks must be chosen accordingly, the new front-end must be compatible with the existing APIs. Examining each candidate from different perspectives is important for the success of the project.

Points are given between 1 and 5. Each candidate was examined on:

- **Maturity** – the selected framework should be actively maintained and documented. Also, the availability of literature and deployed production applications were important features in the examination.
- **Community** - community is usually the matter keeping open source projects alive. It needed to be looked at how the ecosystem is doing by looking at GitHub statistics and Stack Overflow questions (Total, answered and unanswered)
- **General suitability** - finding the right tools for developers is the most important aspect, are they flexible? Will they fit the team? Keeping the code base maintainable by modularizing and templating is as important as is the size of the deployed framework including dependencies.

The architecture of the application was the first to be chosen, as everything would be built around it. There were three frameworks that narrowed the search down to AngularJS, Backbone.js and Ember.js.

AngularJS is open-source and maintained by Google and the AngularJS community and is based on MVC. Its philosophy is to extend the HTML vocabulary, where 2-way data binding and dependency injection relieves developers from writing boilerplate code. It aims to simplify the development process by concurring developers with a high level of abstraction, however, at a cost of flexibility. “Angular was built with the CRUD application in mind” (AngularJS, n.d. a), so it might not be suitable for all use cases.

Backbone.js is an open-sourced component of DocumentCloud, its architecture is MV*. Its philosophy is to provide a minimal set of data-structuring and user interface primitives to help build web applications. It aims to give developers total freedom in the design of their application. As developers have to write lots of boilerplate code and implement the application architecture themselves, it will fit to different kinds of use cases.

Ember.js is open-source and maintained by Yehuda Katz and Tom Dale, its architecture is MVC. It focuses on building scalable desktop-like applications. It relies on front-end templates and depends on the template library Handlebars. It also features 2-way data binding and automatic application state management. It was not designed with an application type in mind, therefore it will also be fit to different kinds of use cases.

3.1.2 Maturity

AngularJS is documented very well. AngularJS’s documentation site contains tutorials and videos by the AngularJS development team and plenty of examples. There is a great deal of literature about AngularJS and Angular is actively maintained. It has been used in production at Google and it is growing in popularity, companies such as NASA and MSNBC are using AngularJS.

Backbone.js is thoroughly documented and there is plenty of free material and literature to learn from. Backbone.js is actively maintained, mainly bug fixes as the API have not been changed in a while. It is used in production for many popular sites like Reddit, Bitbucket, Soundcloud, Tumblr and Pinterest.

Ember.js API-documentation is extensive and the getting started has a great step-by step guide in building an example application. Ember.js is actively maintained, and new versions are released regularly. Ember.js APIs have changed slightly recently, with some breaking changes; thus, updates may be problematic and some external resources may be outdated. Ember.js is also used in production for many sites like ZenDesk, Vine, Twitch.tv and Groupon.

There is no clear winner here, Backbone.js gets most points (see Table *1*) because of the API-stability and abundance of resources.

Table 1. Scores for framework maturity

Framework	AngularJS	Backbone.js	Ember.js
Score	4	5	4

3.1.3 Community

AngularJS had 3 013 watchers on GitHub. There were 132 authors in a period of a month (GitHub, 2014a). AngularJS had the most activity on Stack Overflow, with 64 000 overall questions. 52 764 of them have at least one answer and 10 927 were without any (Stack Overflow, 2014a).

Backbone.js had 1 495 watchers. There were 19 authors in a period of a month (GitHub, 2014b). Backbone.js had a total of 16 704 questions on Stack Overflow with 14 817 answered questions, 1 887 were unanswered (Stack Overflow, 2014b).

EmberJS had 990 watchers on GitHub. There were 49 author in a period of a month (GitHub, 2014c). EmberJS had a total of 12 595 questions with 10 830 answered questions, 1 765 were unanswered (Stack Overflow, 2014c).

In light of the numbers AngularJS gets the highest score (see Table *2*). Even if the numbers are against Backbone.js and Ember.js, they have very active communities and are in no risk of being abandoned.

Table 2. Scores for framework community

Framework	AngularJS	Backbone.js	Ember.js
Score	5	4	4

3.1.4 General suitability

AngularJS is a large and complex framework and after the basics its learning curve is quite steep. AngularJS is somewhat opinionated how the application should be structured, which might bring problems during development. AngularJS is not dependent on any other libraries and its minified size is 107 KB (40 KB gzipped). The views in AngularJS are declared in HTML, no separate template-engine is needed. “AngularJS is emphasized in testing and dependency injection” (AngularJS n.d. b), which reduces the amount of deployed errors as they are caught in unit tests.

Backbone.js is a simple and unopinionated framework, and learning its ins and outs is easy. As Backbone.js is unopinionated, there will be no risks in using the client’s existing data structure. Its only hard dependency is Underscore.js, which contains a template engine. In overall it is the smallest of the three frameworks considered. Its minified size without dependencies is 20 KB and with dependencies included 36 KB (12 KB gzipped). There is no strict rules for what libraries can be used with Backbone.js, so a set of tools preferred by the developer can be used.

EmberJS is also a large and complex framework and after some invested time the learning curve becomes less steep. EmberJS is highly opinionated on how the application should be built, which might cause problems during development. EmberJS is the largest of the three, no hard dependencies and weighing in at 345 KB (99 KB gzipped). EmberJS uses the Handlebars templating library, which has a lot of resources and support available.

The opinionatedness of AngularJS and Ember.js sets them back in this comparison. The size gives Backbone.js additional leverage over the others, as it is significantly smaller than Ember.js and less than a half of AngularJS. All in all, Backbone.js was a clear winner (see Table 3).

Table 3. Scores for framework general suitability

Framework	AngularJS	Backbone.js	Ember.js
Score	3	5	3

3.1.5 Summary of considered MVC frameworks

In overall the three considered frameworks were quite equal in points, with no clear winner until examining the general suitability for the project at hand. As AngularJS and EmberJS are somewhat opinionated how the application should be built, the risk of “fighting against” the framework in some part of the project is one that could not be taken, even if it means writing some extra code. Backbone.js was the most suitable for this project, as a lightweight system with the wanted structure could be built. By thorough planning common pitfalls in building applications with Backbone.js can be avoided, such as memory leaks and redundant boilerplate code.

Table 4. Summary of MVC framework comparison

Framework	Maturity	Community	General suitability	Total
AngularJS	4	5	3	12
Backbone.js	5	4	5	14
Ember.js	4	4	3	11

3.2 Selecting the UI framework

Selecting the proper UI framework is important but not as important as the MVC framework. UI frameworks provide developers with tools to make the development faster and easier. We wanted a framework that provides different UI elements like modals and dropdown menus, as well as a responsive grid layout system. Creating a responsive layout enables users to use the application with devices with different resolutions, without problems in the UI breaking. Having a proper framework for UI

development also adds consistency in different views in an application, as well as consistency in the whole range of products in the portfolio of a company. Choosing the wrong framework and having to do UI migration is not as complicated as an architecture migration, however, a task that is very time consuming such as the project will contain over a hundred views in total.

For the UI framework, three options were compared: Bootstrap, Foundation and Semantic UI. The differences between UI frameworks are smaller than MVC frameworks, therefore, comparison was done on a more general level. The important features were as follows:

- Does it have the features we need? If something that is needed is missing, are there third-party solutions available?
- Can a custom build be created with only the needed modules?
- Is the framework project and community active?

Bootstrap was and still is used as the internal style guide at twitter for over a year before it was publicly released (Bootstrap n.d.). The main goal was to provide a set of guidelines to create UI consistency and relieve the maintenance load of multiple different libraries. Bootstrap is currently the most starred repository on GitHub, with 74 866 stars (GitHub 2014d). Bootstrap is also likely the most popular UI-framework at the moment, therefore help and resources are definitely available. It contains plenty of different features, most of what were needed and the ones not included are available as open source plug-ins. The build customization is effortless, it can be done online or locally. The custom build enables the selection of features and changing the appearance like fonts and colors. Development and the community are very active, there are many developers, 42 authors last month in total (GitHub 2014d). The core team consists of 10 people.

Foundation’s origins are in the ZURB style guide, which ZURB used on every client project. Eventually, they decided that they needed a framework that allowed developers to prototype rapidly (Foundation n.d.). Foundation has 18 698 stars on GitHub (GitHub 2014e) and it has plenty of resources on their developer pages and the features missing are available as open-source. As in Bootstrap, the custom build process is easy and it can be done from the browser or locally. Selecting the needed

features and changing colors and fonts is possible. The development and community are not that active; however, bug fixes are fixed and merged in to the master branch from time to time, 35 commits from 27 authors in the past month (GitHub 2014e). The question on the GitHub repository gets answered, however, it seems to take a few days.

Semantic UI is a new player in the UI-framework field as it has been introduced in September 2013 (Semantic-UI, 2014). Semantic UI has 12 300 starts (GitHub, 2014f), which is quite impressive considering its age. Semantic UI tries to unite designers and developers. By leveraging a semantic descriptive language for its classes and naming conventions. When other frameworks use abbreviations, Semantic UI uses real words used in plain English (Gerchev 2014.). Semantic UI is neatly documented with many examples, but other resources are hard to find. The custom build process is done locally with a step-by-step wizard, which makes it easy to use. Semantic UI is actively developed and maintained, mostly it is developed by one person, which is a risk in the continuity of the framework. 234 commits by 9 authors in the last month, 232 of which were made by the main developer (GitHub 2014f).

Making a decision on the UI framework was not as clear as the decision made on the MVC framework. All frameworks have the modules needed except for a date and time picker, which can be found as a third party plugin for all three. Bootstrap and Foundation were considered equal in features, however, Bootstrap was preferred simply because the ecosystem in Bootstrap is huge and the team has previous experience in working with it. Deciding between Bootstrap and Semantic UI instead placed a challenge. Semantic UI's approach in using common language in declaring elements and using plurality in expressing groups is intriguing. The major problem with Semantic UI is that it is a single person project at the moment and relying on a project which might change a great deal when the core team grows or dies altogether is a risk that the author was not willing to take. Semantic UI is a project to watch and possibly use in the future, nevertheless, Bootstrap was the final decision. The key advantages in Bootstrap were the huge community, the available resources and the previous experience working with it.

3.3 Selecting other needed libraries and tools

Backbone.js is a minimal framework that provides the bare essentials and it was determined to bring in some libraries to help development. Module and dependency management might not be an issue in small applications, but without doubt the project needed something more than loading modules with multiple script-tags. Using script-tags to load a large amount of files is risky in many ways. For example, browsers load script tags synchronously, which means the page will not be rendered before everything is loaded. Dependency management is also challenge, as the scripts with implicit dependencies have to be loaded in the right order. RequireJS is a file and module loader designed to do this and more; it loads scripts asynchronously and in the right order. For managing templates stored in plain text format a RequireJS text-plugin needed to be included.

While Underscore.js provides a templating engine, it was decided to use Handlebars instead. Handlebars is a logic less templating language which forces logic and presentation separation and its syntax keeps templates easy to read. Handlebars also escape all expressions thus being an addition in security against XSS-attacks.

jQuery is not a dependency of Backbone.js, however, Bootstrap's JavaScript plugins were planned to be used which require jQuery. jQuery also provides easy and consistent cross-browser DOM manipulation, AJAX requests and event handling.

For automation Grunt was chosen. The less work a developer has to do when performing repetitive tasks like minification, compilation, unit testing and linting, the easier their job becomes (Grunt n.d.). In this project, Grunt automates the deployment process by checking the source code for errors, by minifying the source code and CSS files and copying all needed files into a folder for distribution. The deployment process is discussed more in detail later on in the thesis (chapter 4.7).

JSHint is an open source tool for error and potential problem detecting in JavaScript code and a way to enforce a team's coding conventions (JSHint n.d.). JSHint helps developers to follow the coding conventions defined in the project. Checking the code style with JSHint is a part of the deployment process and the build will not pass if the check fails.

For version control Git was opted for. Git has been used in the previous projects in the company without problems. As well as having previous experience in Git, it has become a crucial part of the development process after Bitbucket was put into use.

4 FEATURE DEVELOPMENT PROCESS

4.1 Feature-driven development

As the team in this project is small a lightweight version of FDD was used with slight modifications to fit the project. The project manager manages a backlog of features that are needed and plans features to be built for two months ahead. In the beginning of each month the project manager and developers have a meeting to review results of the previous month and prioritize the features allocated for that month. The initial allocation is a rough draft to which developers should suggest changes. A suggestion can be made for changing the time estimates or splitting larger features into smaller tasks. Good cooperation between management and developers is very important.

> *Any system where management writes a schedule and hands it off to programmers is doomed to fail. Only the programmer who is going to implement a feature can figure out what steps they will need to take to implement that feature.* (Spolsky 2007).

In these projects the time estimates are divided into five tiers:

1. XS, maximum work load of 0.5 days
2. S, maximum work load of 1 day
3. M, maximum work load of 1.5 days
4. L, maximum work load of 2 days
5. XL, estimated work load over 2 days

Dividing XL-tier tasks into smaller ones is strongly encouraged, as it makes them easier to design and manage which adds productivity. It cannot be stressed enough how important it is to divide a large feature into smaller tasks. “This forces you to actually figure out what you are going to do” (Spolsky 2007). Splitting tasks into smaller sub-tasks is not strictly a software development method, but a project management method in general called work breakdown structure.

> *The elements of the WBS may include an identifiable item of equipment or software, a deliverable data package, an element of logistic support, a human service or a combination of thereof* (Blanchard 2004, 270).

4.2 Example case: price lists

The price list feature that is examined in the following chapters is a good example of this. The principle in the feature is to be able to control listings of prices between companies (see Figure 3), be it material or work. The price list consists of company specific price list rows and have a set of fields like: name, identifier and price. Price lists also have an essential process, the approval process. When the price lists are sent from one company to another or a price has changed, both parties will be informed of the event and it has to be approved before the price list can be used again. The last sub-feature is price list collections, which contain a set of identifiers to enable selecting a set of rows with matching identifiers from a price list which can contain hundreds of rows.

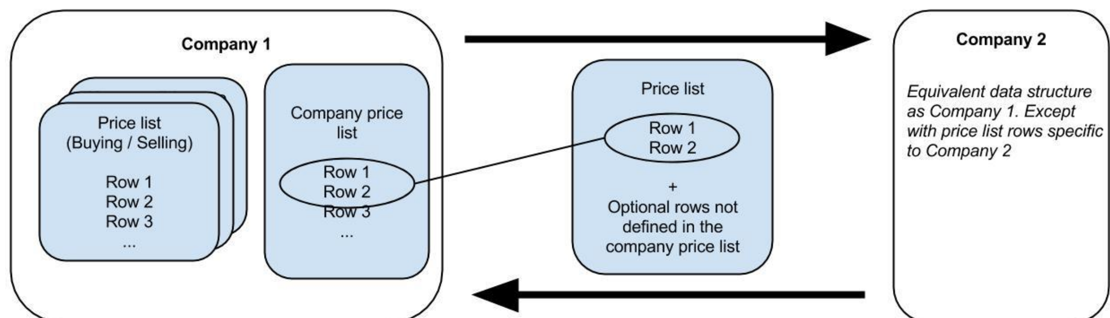

Figure 3. Structure of the price list feature

The feature does not seem that complicated; however, it surely will take more than two days, so it will have to be split into smaller tasks:

1. Create / update / delete company specific rows (M)
2. Create a list of rows and send it to another company (M)
3. Approve and edit the sent / received rows (M)
4. Create price list collections (S)

The feature was split into logical sub-features, which are easier to perceive and manage. A part of the first task, creating company specific price list rows will be used as an example in the chapters 4.2 – 4.4. A summary of the whole feature will be presented in chapter 7.1.

4.3 UI-design and implementation

When the feature is assigned to a developer, the developer and project manager will have a meeting where the feature will be designed in detail and a rough draft of the UI is produced. When the UI draft is ready, the developer then examines the existing features for modules that could be integrated in the new feature. Using existing modules is important because reusing code saves time, and money. Since someone has already solved a problem, there is no point in solving it again (Talk 2014).

After the examination is done a new branch should be created in to the Git repository and the draft should be created in to a mock-up using Bootstrap and the possible existing modules. When the mock-up is completed, it will be reviewed either by the project manager or another developer of the team as for its usability and general visual appearance. By reviewing the interface at an early stage helps to catch inconsistencies and problems in usability. For instance, during the interface review a developer of the team noticed an inconsistency with the rest of the application in the feature design. When something that is quite small is being created in MDO hub, the form is placed in a modal. In the initial mock-up of the price list feature, the form was at the bottom of the UI (see Figure 4).

Figure 4. Inconsistency marked in red. It should be in the group marked in blue.

After the review, the following changes were made to make the UI follow the same patterns as other features use. A button triggering the modal for creating new rows was added into the button group, which already contains controls concerning the rows below (see Figure 5).

Figure 5. Create button added to the button group to remove inconsistency

The form was placed in a modal (see Figure 6) which is consistent with the rest of the system.

Figure 6. The form was moved into a modal to remove the inconsistency.

When the mock-up is considered done, it is time to convert it into a template. The process of converting a static HTML page to a template is quite simple. Basically the static values are replaced with Handlebars expressions and helpers. For example, a stripped version of the HTML used in the modal just created is:

```
<div class="modal-header">
    <button type="button" class="close" data-dismiss="modal" aria-
    hidden="true">×</button>
    <h3>Add new price list row</h3>
</div>

<div class="modal-body row-fluid">
    <div class="span12">
           <label>Identifier:</label>
           <input class="span-block" name="identifier" type="text"
           placeholder="Identifier">
           <label>Name:</label>
           <input class="span-block" name="name" type="text"
           placeholder="Name">
           <label>Price:</label>
           <input class="span-block" name="price" type="number”
           min=”0”>

    </div>
</div>

<div class="modal-footer">
    <button class="btn" data-dismiss="modal" aria-hidden="true">Cancel</button>
    <button class="btn btn-primary addPriceListRow">Add</button>
</div>
```

Bootstrap modals are divided into three parts: the header, the body and the footer. The content of the modal is static, and dynamic content can be easily added with Handlebars. The template of the modal is internationalized by replacing static text with Handlebars expressions:

```
<div class="modal-header">
     <button type="button" class="close" data-dismiss="modal" aria-
hidden="true">×</button>
     <h3>{{text.addPriceListRow}}</h3>
</div>
<div class="modal-body row-fluid">
     <div class="span12">
             <label>{{text.identifier}}:</label>
             <input class="span-block rowAttribute" name="identifier" type="text"
             placeholder="{{text.identifier}}">
             <label>{{text.name}}:</label>
             <input class="span-block rowAttribute" name="name" type="text"
             placeholder="{{text.name}}">
             <label>{{text.price}}:</label>
             <input class="span-block rowAttribute" name="price" type="number”
             min=”0”>
     </div>
</div>
<div class="modal-footer">
             <button class="btn" data-dismiss="modal" aria-
             hidden="true">{{text.cancel}}</button>
             <button class="btn btn-primary
             addPriceListRow">{{text.add}}</button>
</div>
```

Handlebars compiles the template in to a function where the expressions are represented as variables given to Handlebars when executing the function. Compiling and calling the template is easy. In the example the template of the modal is initiated in the modalTemplate variable:

```
var modalTemplate = Handlebars.compile(modalTemplate);
$('#Modal').html(modalTemplate({
             text: {
                    addPriceListRow: MDO.localization.getText('addPriceListRow'),
                    identifier: MDO.localization.getText('identifier'),
                    name: MDO.localization.getText('name'),
                    price: MDO.localization.getText('price'),
                    cancel: MDO.localization.getText('cancel'),
                    add: MDO.localization.getText('add')
             }
}));
```

The text that will be placed in the modal is fetched from the localization model, which contains the translations for all of the UI-elements. Placing the modal template in the element would produce a modal like in Figure 6, with translated content based on the language setting. Pages with a great amount of dynamic content are easier to update during the application execution, as the function takes all variables and returns the evaluated HTML instead of having to change the values one by one. After the template is done and the translations written the developer can continue in to creating the business logic of the feature.

4.4 Front-end implementation

4.4.1 Model implementation

“Backbone models contain data for an application as well as the logic around this data” (Osmani 2013, 28). The model does not have to initiate the attributes it will contain, Backbone will do that automatically. Instead default values can be defined for the attributes, and doing so is a good practice as the model represents the API. It is possible to use a Backbone model without extending it, if nothing else than setting and getting attributes is needed, however, in most cases extending the Backbone.Model class is the way to go. Defining a model and setting default values for a price list row-model is easy:

```
var PriceListRow = Backbone.Model.extend({
    defaults: {
            identifier: “”,
            name: “”,
            price: 0
    }
});
```

After this every row created will have the defined default attributes, if not specified otherwise. Getting and setting data is done by calling the set or get function of a model:

```
var row = new PriceListRow();
alert(row.get(”price”));
row.set(”name”, ”Welding”);
alert(row.get(”name”));
alert(row.get(“test”));
```

The first alert will bring up ”0”, as it is the default value for the price attribute and the second will alert bring up ”Welding”. The last alert will bring up “undefined” because it was not declared in the defaults nor was it set afterwards. Another useful feature in models is validation, which allows the model to check if its attributes are set correctly. It can be used before setting an attribute or saving the model to the back-end. Validation is important for the user experience when you can catch input errors without extra HTTP requests. A basic validation for the price list model could be:

```
var PriceListRow = Backbone.Model.extend({
    defaults: {
        identifier: “”,
        name: “”,
        price: 0
    },
    validate: function (attributes) {
        if (typeof attributes.name !== 'string' || !(attributes.name
        instanceof String)) {
            alert(“Name should be a string”);
        }

        if (isNaN(attributes.price)) {
            alert(“Price should be a number”);
        }

    }
});

var row = new PriceListRow();
row.validate();

row.set(”price”, “twenty”);
row.validate();
```

When calling the validate function on the row model without setting anything will cause no alerts, as everything is as it should. The second validate call will trigger an alert as the price attribute should always be a number. Displaying alerts in a production application is discouraged and should be avoided. The solution is to create a member variable for the model in which the errors are stored in and handled by the view.

As validating data is important in the front-end, it is extremely important to remember that JavaScript code can be modified while the application is running, so one cannot truly rely on the safety of the validation. Using validation for enhancing the user experience is a great idea, however, relying only on front-end validation is very risky if the data is being stored in a database. The validation logic will have to be implemented into the back-end as well.

> *Validation routines are beneficial on the client side but are not intended to provide a security feature as all data accessible on the client side is modifiable by a malicious user or attacker* (Auger & Shenoy 2009).

4.4.2 View implementation

In its simplest form, a view passes the data from the model to the UI and back. Creating views with Backbone.js is simple, the company price list shall be used as an example. The view is simple (see Figure *7*), the user can add new price list rows, edit

or remove existing ones and navigate to the price lists or price list collections view.

All	Name	Description	Price	VAT	Unit
	T1 Laitetyömaan perustaminen, Etelä-Suomen lääni		50	24	
	T2 Laitetyömaan perustaminen, Itä- ja Länsi-Suomen, sekä Oulun lääni		70	24	
	T3 Laitetyömaan perustaminen, Lapin lääni		55	24	
	T12 Antennikaapelien asennus mastoon ½" ja 7/8 m		85	24	
	T11 Lisä TTRX:n asennus		120	24	
	T13 Antennikaapelin asennus kiinteistöön ½" ja 7/8 jm		75	24	h
	T16 Antennin suuntaus		50	24	h
	Poltin		50	24	kpl
	Kilometrikorvaus		0.43	0	km
	Öljykattilan asennus		150	24	kpl

Figure 7. The company price list view

The list is a separate module, which is used across the application called ResponsiveList. It lists specified attributes of a collection of models and enables sorting and changing the values that are displayed in a single column. The views in Backbone.js are defined similarly as Models:

```
var CompanyPriceList = Backbone.View.extend({
    companyPriceListTemplate: utils.template(CompanyPriceListTemplate),
    initialize: function () {
                  this.subViews = {};
                  this.subViews.rowList = new ResponsiveList();
                  this.subViews.createRowModal = new CreateRowModal();
                  this.model = new CompanyPriceList();
    },

    render: function () {
           var templateData = {
                  text: getTranslations()
               };

           this.$el.html(this.companyPriceListTemplate(templateData));
           this.$(“#RowList”).html(this.subViews.rowList.el);
           this.$el.append(this.subViews.createRowModal.el);

           this.model.fetchPriceList().done(this.updateList);
           return this;
    },

    updateList: function () {
           this.subViews.rowList.updateItems(this.model.rows);
    }
});
```

The frame of the view is not complicated. The companyPriceListTemplate is a Handlebars function ready to be invoked. Initialize is called when the view is constructed, an object for sub-views is created for easy management. The

ResponsiveList sub-view is the general list module mentioned earlier and the CreateRowModal sub-view is a view containing a modal for creating price list rows. A model is also attached to the view, so the data is always accessible in the view. The render function is called when the view is to be placed on to the DOM. In this example the templateData variable only contains the translations for the view. After the translations are fetched the template is invoked and its result placed in to the views element. The list module's element is then placed into the DOM, each view has a DOM element, and it is stored in the el property. The $el property is a jQuery object of the view's element. As each view controls its element, the list module can manipulate its element freely and the manipulations will affect the DOM. When the DOM is setup properly, the model may fetch data from the server. When the model is done fetching, it will parse the results and invoke the updateList-function. The function will then pass the rows from the model to the list module which will display them in the DOM.

Listening for DOM events is one of the key tasks in views. Defining which events are to be listened and what function is responsible of handling them is simple, in addition to the previously introduced functionality the events object and the callback functions are defined:

```
var CompanyPriceList = Backbone.View.extend({
    events: {
        “click .showCreateRowModal”: “showCreateRowModalHandler”,
        “click .createPriceListRow”: “createPriceListRowHandler”
    },

    /* Functionality defined in previous example */

    showCreateRowModalHandler: function () {
        this.subViews.createRowModal.show();
    },

    createPriceListRowHandler: function () {
        var self = this;
        this.model.addRow(this.subViews.createRowModal.model)
        .done(function() {
            self.subViews.createRowModal.reset();
            self.updateList();
        });
    }
});
```

The events object contains key value pairs of selectors and callbacks. The first part defines what DOM Event to listen to and the second part is the jQuery selector to focus the listening on to something. In this example two buttons are being listened for click events. As the user clicks the “create new row” button (visible in Figure *7*),

the view catches the event and invokes the handler which invokes the modal to activate and slide down. The createRowModal view will construct a new model and use the model's validate method that was described in chapter 4.3.1 to validate the inputs. The button for creating the row will remain disabled until the model passes validation (see Figure 8). Attributes that are invalid will be marked with a red ribbon on the left side of the input (see Figure 8).

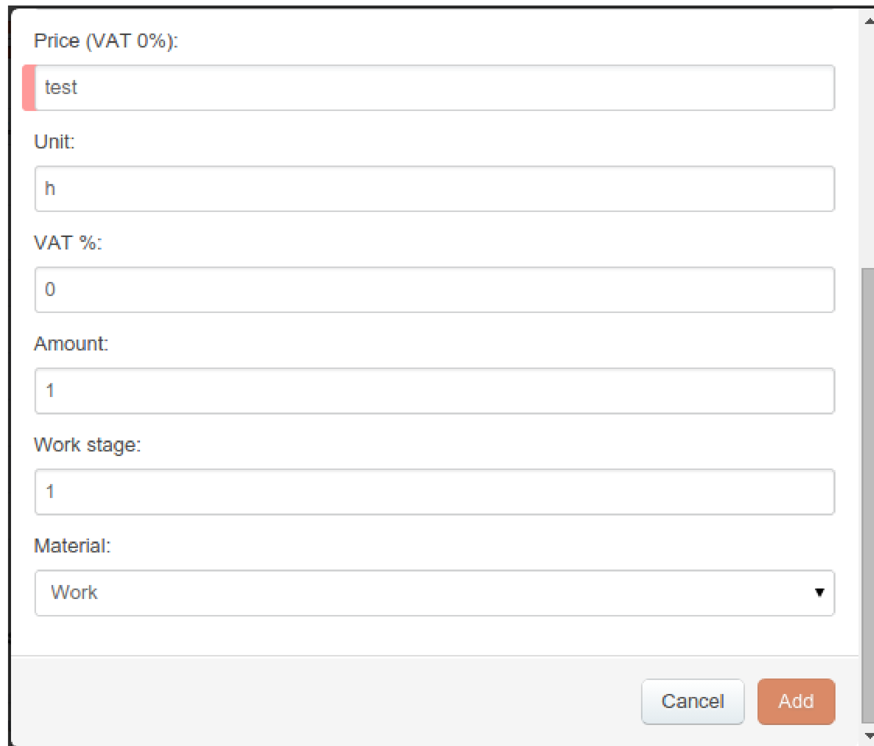

Figure 8. Add row modal with invalid attribute

If the model passes validation, the add button will be enabled and the user can add the new row to the list. The CompanyPriceList model's addRow function makes an AJAX request to the back end and the back end will save the data to the database. The request and response will be described in chapter 4.4. After the AJAX request is done, the model will be added to the collection and the list will be updated to contain the new row. Reset will be invoked on the CreateRowModal view, which will hide the modal.

4.5 Back-end specification

The default attributes in the defined models represent the database columns and should be added into the database schema. Often, as presented in the price list row example they are primitive types and can be stored in a single table.

The addRow function mentioned in 4.3.2 invokes a jQuery post method and returns the Promise object created by the post method. A Promise is an object that represents a one-time event, typically the outcome of an asynchronous task like an AJAX call. At first, a Promise is in a pending state. Eventually, the Promise is either resolved (meaning the task is done) or rejected (if the task failed) (Burnham, 2012). Callback functions can be attached to the Promise, like in chapter 4.3.1. After the AJAX call is completed the attached callbacks will be invoked, done will be invoked if the call was completed with an HTTP status code between 200 and 299 or with an HTTP status code 304, otherwise the fail callback will be invoked. The addRow function is defined in the CompanyPriceList model like this:

```
var CompanyPriceListModel = Backbone.Model.extend({
    /* Defaults and validation logic */
    addRow: function (model) {
            var self = this;
            return $.post(MDO.backendURL, {
                    action: “createPriceListRow”,
                    data: model.toJSON()
            }).done(function () {
                    self.rows.add(model);
            });
    }
});
```

The addRow function also adds a done callback for the post request. The data is a model converted to JSON by invoking the toJSON method, which gathers all of the model’s attributes and creates and JSON object with the data. The callbacks are executed in the order they were added, so the new row will be added to the collection before the done callback is invoked in the view.

4.6 Feature inspection

When the feature has been done, a pull request will be created in the repository. This lets everybody involved know that they need to review the code and merge it into the master branch. However, the pull request is more than just a notification—it is a

dedicated forum for discussing the proposed feature. If there are any problems with the changes, team mates can post feedback in the pull request and even tweak the feature by pushing follow-up commits. All of this activity is tracked directly inside of the pull request. (Atlassian, n.d.)

The code review process itself is simple, the pull request will be presented in Bitbucket in a user friendly web interface (see Figure 9).

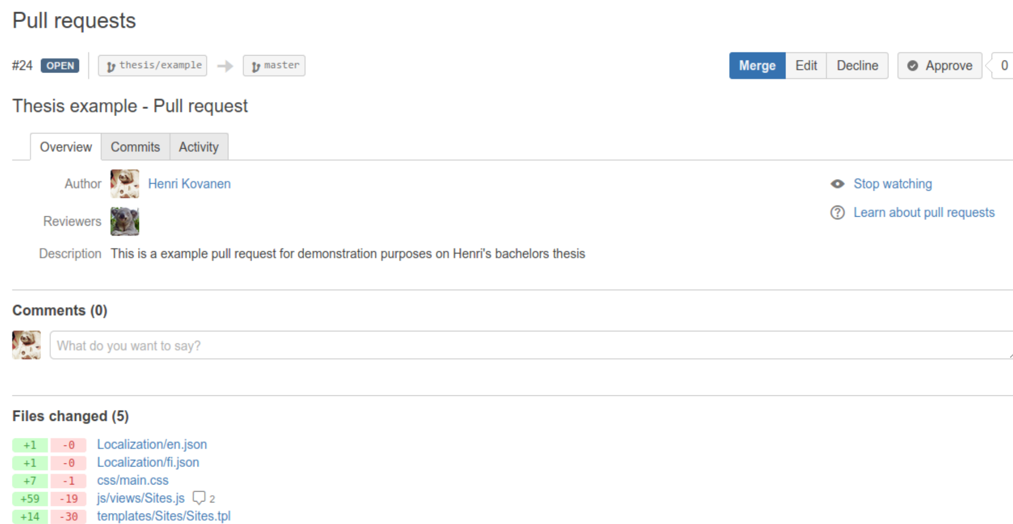

Figure 9. Bitbucket's pull request interface

A set of coding guidelines has been set to help the code readability and maintenance. Instead of each developer coding in their own preferred style, they will write all code to the standards outlined in the document. This makes sure that a large project is coded in a consistent style — parts are not written differently by different programmers. Not only does this solution make the code easier to understand, it also ensures that any developer who looks at the code will know what to expect throughout the entire application (Mytton 2007). The reviewer will check the code style and that the code is properly commented. The interface enables the whole team to collaborate and review the pull request, comments can be added in general (see Figure *9*) or anywhere in the code (see Figure *10*).

Figure 10. Example of a code comment in Bitbucket

The testing is done by pulling the branch to the reviewer's local repository and going through the functionality of the feature. If any bugs or other things that need changes are found, they are usually reported in the pull request and fixed by the author. Any follow-up commits will be automatically added to the original pull request (Atlassian, n.d.). As well as functionality testing, the code should be inspected for syntax errors and deviations from the coding guidelines by running the JSHint-grunt task, which is described later in the deployment section. When the pull request is reviewed and considered done, it can be merged to the master branch. The web interface provides a way to merge and close the branch easily. After the pull request is merged it is time to deploy the new feature.

4.7 Deployment

The deployment process is automated by Grunt tasks. When a feature is ready for deployment all the developer needs to do is to run the Grunt build task, which triggers the needed procedures to create a new build for distribution. While automation makes the deployment faster and brings consistency as developers do not have to run the tasks manually. The process is not overly complicated, but it is easy to forget a step resulting in an unusable build.

The first task to be run is a JSHint task which checks the code for syntax and style errors, if there is any the build will not pass (see Figure 11).

```
khenri at localhost in ~/public_html/mdo-hub on thesis!
$ grunt
Running "jshint:build" (jshint) task

js/views/Sites/BulkEdit.js
  line 489  col 54  Missing semicolon.

  ⚠  1 warning

Warning: Task "jshint:build" failed. Use --force to continue.

Aborted due to warnings.
```

Figure 11. A failed grunt process

Failed tasks can be forced to skip warnings, however, it is strongly discouraged as the warnings are there for a reason and skipping them will break oncoming builds.

The second task is a clean task which removes all of the old build files from the distribution folder.

The third task is CSS-minification, where all of the used CSS files are combined and minified into one file. The basic idea in minifying is to remove from the source code all characters that are unnecessary and merge the files into one, which reduces HTTP-request pollution and file size. For preventing browsers to cache an old build, each new minified file will have a new unique name, in the thesis project the minified files were coded to contain an identifier part and a unique part. The unique part is the epoch time when the build task is run, so the file name for a minified CSS file could be for example: MDO-1417009517.min.css.

The fourth task is JavaScript minification, using the RequireJS optimizer. The optimizer combines scripts that are related to each other into build layers and minifies them with UglifyJS (RequireJS, n.d.). Skipping errors in the JSHint task may cause problems in the minification process and produce an unusable build.

The fifth and final task is a copy task, which copies all of the needed resources to the distribution folder. Most of the resources are images and fonts. Copying and modifying the index.html to point to the newly build CSS and JavaScript files.

5 CONCLUSION

5.1 Results

The objectives of the thesis was to compare and select open source tools for a front-end development stack that can be used in MDO hub and future projects and to implement a price list feature with the chosen tools using existing processes used in the project.

The comparison part resulted in the selection of multiple tools which complement each other. Backbone.js was selected from the MVC frameworks to provide application structure, but Backbone.js itself is not enough as it does not provide dependency management. For dependency management RequireJS was chosen, as there was plenty of available documentation on how to use RequireJS with Backbone.js. The only hard dependency of Backbone.js is Underscore.js, and it provides a simple templating engine, but Handlebars was chosen to be used instead. The UI framework chosen was Bootstrap because of its popularity and the team had previous experience working with it. jQuery was selected because Bootstrap modules depend on it and jQuery also provides event handling and AJAX functions for the whole application. Grunt with a set of extensions was chosen for task automation.

As a concrete result of the comparison part of the thesis, the previously mentioned frameworks and tools were combined for a full front-end development stack. The stack is very versatile and can be used in many different projects and it adds consistency and a good base to start from in all of the client’s future projects that are built with the stack.

The price list feature was partly covered in chapter 4 and the remaining parts will be summarized in this chapter. In addition to the create control the price lists main view enables users to remove and modify existing rows. Removing rows is simple, user selects one or multiple rows by clicking the checkboxes on the list. After the selection is done, user clicks the remove button and the rows are removed. To modify a row or rows the user must click on the “Quick-edit” button and the list module will replace the text elements with input elements. The changed attributes will be updated to the back-end instantly by the model.

Creating and sending a price list to another company is done from the price lists view. Price lists view contains a similar list, but instead of a single price list row, one row in the list represents a price list that has been sent or received. As the price lists have an expiration date, the view contains a button to show the outdated price lists. The price list view contains multiple controls, the user can view and add comments to the price list using a simple text area. The user can change the valid date of the price list, by using an intuitive date picker. Removing and editing existing rows is similar to the company price list view, the user clicks on "Quick-edit" and the text elements are replaced with text inputs. If changes are made to the price list, the user can revert the changes that were made or send the price list for approval. The made changes are highlighted with an asterisk so the receiving user can see the changes easily. New rows can be added from the company price list or rows specific to a single price list can be created in a similar way as in the company price list. An existing price list can be used as a template in creating a new price list, so users can send a similar price list to multiple recipients. If the price list is approved by the recipient new invoices can be created based on the price list.

The price list collections view contains a list of collections, from which the user can open or remove collections. The opened collection view contains a list of rows. The user can remove existing rows or add new ones. The price list rows in the collection differ from the rows used elsewhere, as they only contain an identifier, a name and an amount.

The feature was implemented as a part of this thesis and the feature has been deployed in to production and has been used actively. The implementation of the feature was successful. Developing the feature with the process used in the project inconsistencies were caught like intended and the UI was to the client's liking. The look and feel was designed so that the feature is in line with the rest of the application and it fits in perfectly. The implementation stage was a learning process and some of the parts needed to be refactored and enhanced, but no major rewrites were required, which gave the writer good insight on how the tools of the development stack work and should be used. The implementation decisions and patterns used in the feature have been discussed with the team working on the project and many of them have been adopted to many other features implemented.

It is easy to say that while implementing and gathering information from various resources on how to implement features with the development stack, the writer and the whole team gained knowledge that can be considered valuable.

5.2 Analysis and project continuation

The objectives of the thesis were to research and select a set of frameworks and tools for front-end development of the MDO hub application and to implement a price list feature to the application. It was also requested that the selected tools could be used in future projects by the client.

With the research made in the thesis the client has an insight in the popular open source front-end frameworks and tools. Based on the selection criteria the most suitable tools for the client were selected. If the schedule would have allowed it, a more in depth analysis and a hands on test of the MVC frameworks would have been useful, as it would have given more insight in the general suitability of the frameworks. The selection is subjective and cannot be used generally as some parts of the application suite were already in production and the tools had to be compatible with them.

Creating a SPA instead of a traditional web application has proven to be a right choice, and the application has been in active development since the development stack was created. Members of the development team have been introduced to the development stack and it has proven to be very versatile and powerful. One fear was that integrating the new tools to an existing development process might be a difficult task, but the integration caused no problems at all. The flexibility and multipurpose use possibilities are in line of the request that the tools can be used in future projects as well.

Unit testing is missing from the selected tools and the extensibility of the stack was proven when QUnit was introduced to the stack by another developer. The development stack is not complete, and it might never be. I think the possibility to add new tools easily is very important and enables the development stack to evolve, when needed. From my perspective the selected tools were right for the project and the objective is fulfilled.

The objective of creating a price list feature using the tools and predefined development process went better than excepted. The new tools were used and integrated with the existing process without problems. As a result the feature was completed and has been distributed to the customers of Versine Ltd.

There was a lot to learn during the writing of the thesis. JavaScript MVC frameworks and automated tasks were a completely new thing for the author. The writing and research process gave the author very good knowledge in SPA development in general and specifically with Backbone.js.

The web application development is constantly changing, what might be powerful and efficient now might be outdated in six months. Keeping tabs on the current events of the developer community helps being on the frontier of new technologies. It seems like Ember.js is currently gaining a lot of buzz and popularity with new features, while Backbone.js receives regular bug fixes and a possible major version bump in the distant future. Does that make Ember.js a better framework in general? There is a job for both, and when the right tools find the right job happy developers are made and they will produce a good product.

REFERENCES

Aho 2014. Intranet message 19.5.2014. Versine Ltd.'s Chairman of the Board's description of the company structure.

AngularJS, n.d. a. Angular developer guide, introduction Accessed on 16.11.2014. Retrieved from https://docs.angularjs.org/guide/introduction

AngularJS, n.d. b. Angular developer guide, unit testing. Accessed on 16.11.2014. Retrieved from https://docs.angularjs.org/guide/unit-testing

Atlassian, n.d. Making a Pull Request. Accessed on 1.12.2014. Retrieved from https://www.atlassian.com/git/tutorials/making-a-pull-request/example

Auger & Shenoy, 2009. Improper Input Handling. Accessed on 1.12.2014. Retrieved from http://projects.webappsec.org/w/page/13246933/Improper%20Input%20Handling

Blanchard, 2004. System Engineering Management. 3rd ed. Hoboken: John Wiley & Sons.

Burnham, 2012. Wrangle Async Tasks With JQuery Promises. Accessed on 5.12.2014. Retrieved from http://code.tutsplus.com/tutorials/wrangle-async-tasks-with-jquery-promises--net-24135

Foundation, n.d. About Foundation. Accessed on 29.11.2014. Retrieved from http://foundation.zurb.com/learn/about.html

Gerchev, 2014. Introducing: Semantic UI Component Library. Accessed on 29.11.2014. Retrieved from http://www.sitepoint.com/introducing-semantic-ui-component-library/

GitHub, 2014a. AngularJS monthly Pulse. Accessed on 16.11.2014. Retrieved from https://github.com/angular/angular/pulse/monthly

GitHub, 2014b. Backbone.js monthly Pulse. Accessed on 16.11.2014. Retrieved from https://github.com/jashkenas/backbone/pulse/monthly

GitHub, 2014c. Ember.js monthly Pulse. Accessed on 16.11.2014. Retrieved from https://github.com/emberjs/ember.js/pulse/monthly

GitHub, 2014d. Bootstrap monthly Pulse. Accessed on 1.12.2014. Retrieved from https://github.com/twbs/bootstrap/pulse/monthly

GitHub, 2014e. Foundation monthly Pulse. Accessed on 1.12.2014. Retrieved from https://github.com/zurb/foundation/pulse/monthly

GitHub, 2014f. Semantic UI monthly pulse. Accessed on 1.12.2014. Retrieved from https://github.com/Semantic-Org/Semantic-UI/pulse/monthly

Grunt, n.d. Grunt introduction. Accessed on 20.11.2014. Retrieved from http://gruntjs.com/

Mikowski & Powell, 2013. Single Page Web Applications. Shelter Island:

Mytton, 2004. Why you need coding standards. Accessed on 27.11.2014. Retrieved from http://www.sitepoint.com/coding-standards/

Nielsen, 1993. Response Times: The 3 Important Limits. Accessed on 10.11.2014. Retrieved from http://www.nngroup.com/articles/response-times-3-important-limits/

Osmani, 2012. Understanding MVC And MVP (For JavaScript And Backbone Developers). Accessed on 20.11.2014. Retrieved from http://addyosmani.com/blog/understanding-mvc-and-mvp-for-javascript-and-backbone-developers/

Osmani, 2013. Developing Backbone.js Applications. Sebastopol: O'Reilly Media.

RequireJS, n.d. RequireJS optimizer documentation. Accessed on 16.11.2014. Retrieved from http://requirejs.org/docs/optimization.html

Semantic-UI, 2014. Release notes. Accessed on 29.11.2014. Retrieved from https://github.com/Semantic-Org/Semantic-UI/blob/master/RELEASE-NOTES.md

Spolsky, 2007. Evidence Based Scheduling. Accessed on 20.11.2014. Retrieved from http://www.joelonsoftware.com/items/2007/10/26.html

Stack overflow, 2014a. Newest AngularJS questions. Accessed on 16.11.2014. Retrieved from http://stackoverflow.com/questions/tagged/angularjs

Stack overflow, 2014b. Newest Backbone.js questions. Accessed on 16.11.2014. Retrieved from http://stackoverflow.com/questions/tagged/backbone.js

Stack overflow, 2014c. Newest Ember.js questions. Accessed on 16.11.2014. Retrieved from http://stackoverflow.com/questions/tagged/ember.js

Talk, 2014. DocForge - Code reuse. Accessed on 16.11.2014. Retrieved from http://docforge.com/wiki/Code_reuse