jamk.fi

Single-page Application Frameworks in Enterprise Software Development

Juha Kokkonen

Master’s thesis
May 2015

Master's Degree Programme in Information Technology
School of Technology, Communication and Transport

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Description

Author(s)	Type of publication	Date
Kokkonen, Juha	Master's thesis	15.05.2015
		Language of publication: English
	Number of pages: 44	Permission for web publication: x

Title of publication

Single-page Application Frameworks in Enterprise Software Development

Degree programme

Master's Degree Programme in Information Technology

Tutor(s)

Huotari, Jouni
Rintamäki, Marko

Assigned by

Fifth Element Oy

Abstract

The primary goal of this thesis was to assess the feasibility of JavaScript single-page application frameworks in the scope of enterprise application development. Two popular single-page application frameworks were studied in this thesis: Backbone.js and AngularJS.

The secondary goal of this thesis was to build documentation for developers who are moving from one framework to another and are seeking to learn how to map the common concepts among the selected frameworks.

The results of this study indicate that it is often feasible for single-page application frameworks to be integrated into enterprise applications. However, while the single-page architecture moves many responsibilities of an application from the server to the the client, it is crucial that the data access to enterprise services is organized in web client friendly-manner. Other technologies are required to convert and proxy the enterprise services that utilize protocols and data formats that are beyond the capabilities of a web browser.

Application requirements should be carefully considered when choosing the framework. Backbone.js provides a development framework with a minimalistic approach. AngularJS takes a more holistic approach to what features a single-page framework should provide. If the broader feature set of AngularJS can be fully utilized in the application development, it probably offers better productivity. On the other hand, if the application requires noticeably different behavior, it is easier to modify Backbone.js to suit these requirements.

Keywords/tags (subjects)

JavaScript single-page application frameworks, Enterprise application development, AngularJS, Backbone.js

Miscellaneous

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Kokkonen, Juha	Opinnäytetyö	15.05.2015
	Sivumäärä 44	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty: x

Työn nimi

Single-page Application Frameworks in Enterprise Software Development

Koulutusohjelma

Master's Degree Programme in Information Technology

Työn ohjaaja(t)

Jouni Huotari
Marko Rintamäki

Toimeksiantaja(t)

Fifth Element Oy

Tiivistelmä

Opinnäytetyön ensisijainen tavoite oli tutkia JavaScript single-page application-sovelluskehysten soveltuvuutta yritysohjelmistojen kehittämiseen. Opinnäytetyössä vertailtiin kahta suosittua sovelluskehystä: Backbone.js ja AngularJS.

Toissijainen tavoite oli luoda dokumentaatio sovelluskehittäjille, jotka työssään siirtyvät sovelluskehyksestä toiseen ja haluavat yleiskuvan, kuinka vastaavat kokonaisuudet on toteutettu vertailluissa sovelluskehyksissä.

Tutkimuksen tuloksena havaittiin, että single page application -sovelluskehykset ovat useissa tilanteissa käyttökelpoisia yritysohjelmistojen kehityksessä. Vaikka single-page-arkkitehtuuri siirtää monia sovelluksen toiminnallisuuksia palvelimelta selaimeen, on tärkeää, että sovelluksen tarvitsemat palvelut ja tietovarastot ovat saatavilla web-selaimen tukemien protokollien ja tietomuotojen rajoissa. Sellaiset palvelut, joita single-page-sovellus ei pysty suoraan hyödyntämään, täytyy muuntaa selainsovellukselle yhteensopivaan muotoon muiden teknologioiden avulla.

Sovelluskehystä valittaessa tulee kiinnittää huomiota toteutettavan sovelluksen vaatimuksiin. Backbone.js tarjoaa minimalistisen lähtökohdan single-page -sovellusten rakentamiseen. AngularJS pyrkii tarjoamaan kokonaisvaltaisemman näkemyksen siitä, mitä ominaisuuksia sovelluskehyksen tulisi tarjota sovelluskehityksen tueksi. Mikäli AngularJS:n monipuolisempaa ominaisuuskirjoa pystytään hyödyntämään laajasti, on se todennäköisesti tuottavampi valinta. Jos taas sovellus poikkeaa huomattavasti tavanomaisesta, on Backbone.js helpompi sovittaa näihin vaatimuksiin.

Avainsanat (asiasanat)

JavaScript single-page application sovelluskehykset, yritysohjelmistojen kehitys, AngularJS, Backbone.js

Muut tiedot

Contents

1 Introduction....8
1.1 Thesis Objectives....8
1.2 Scope....8
2 Motivation for JavaScript SPA in enterprise application development....9
2.1 Traditional web applications....9
2.2 Single-page web applications....10
2.3 Building blocks of single-page web applications....11
2.4 Choice of platforms....12
3 Accessing enterprise back-end systems....13
3.1 XML....13
3.2 SOAP....14
3.3 JSON....15
3.4 REST....15
3.5 Databases and legacy systems....16
4 Modularity and dependency management....16
5 Build process and application deployment....17
6 Data binding and validation....18
6.1 Templates....18
6.2 Binding data to views....19
6.3 Validating user input....19
7 Testing....20
7.1 Unit testing....20
7.2 Integration testing....21
7.3 Continuous integration....21
7.4 Testing considerations in single-page applications....22
8 Framework study: Backbone.js....23
8.1 Views....24
8.2 Models....25
8.3 Validation....25

8.4 Events....26
8.5 Collections....27
8.6 Persistence....28
8.7 Routing and History....28
8.8 Module system....29
9 Framework study: AngularJS....29
9.1 Application....30
9.2 Controller....31
9.3 Views and Templates....31
9.4 Scopes and Models....33
9.5 Validation....34
9.6 Services....35
9.7 Dependency Injection....35
9.8 Routing and History....36
9.9 Module system....37
9.10 Testing....37
10 Conclusions....38
References....41
Appendices....43

Terms and Acronyms

Ajax	Asynchronous JavaScript and XML, client-side technologies used to create asynchronous web applications
API	Application Programming Interface
CI	Continuous Integration
CRUD	Create - Read - Update – Delete, basic persistence operations in data driven software application
DOM	Document Object Model, a standardized programming model for querying and modifying HTML document in a browser
ECMAScript	Standardised ISO/IEC 16262 language specification, published by European Computer Manufacturers Association
ERP	Enterprise Resource Planning
HTML	Hyper Text Markup Language, a markup language for building web pages and applications
JS	The JavaScript programming language

JSON	JavaScript Object Notation, a light-weight data interchange format
MVC	Model-View-Controller, a design pattern that defines components and their responsibilities in presenting a user interface
REST	Representational State Transfer, software architecture style for building web services
Singleton object	Object that is restricted to maximum of one instance at a time in a system
SOA	Service-oriented architecture, a design pattern to provide application components as network services using open standard interfaces
SOAP	A protocol specification for exchanging XML structured information by web services
SPA	Web application that utilizes Single-page Application programming model
URL	Uniform resource location, a reference to resource in a computer network

1 Introduction

1.1 Thesis Objectives

The first objective of this thesis is to present the case of utilizing JavaScript single-page application frameworks in the scope of enterprise application development. To accomplish this, two popular SPA frameworks were selected for the study: Backbone.js and AngularJS. The comparison of features, strengths and weaknesses of the frameworks are studied in this thesis.

The second objective is to provide a transition guide for developers who are switching from one framework to another. While both single-page application frameworks selected for study could provide similar end results, they take a very different approach at the software architecture level. This thesis attempts to map the common concepts to help developers move between these frameworks with minimal learning time investment.

1.2 Scope

Both frameworks studied in this thesis have a large number of extensions available. Development projects often supplement the standard framework with extensions that are most suited to the project under development. This study aims to assess the capabilities of each standard framework, leaving any third party extensions to the frameworks beyond the scope of this thesis.

It is also feasible for single-page applications to be packaged and deployed as hybrid desktop or mobile applications. While general issues regarding hybrid applications are included in the discussion, details on how to interface with specific vendor platforms are left out of the scope of the thesis.

2 Motivation for JavaScript SPA in enterprise application development

Single-page web applications aim to provide desktop application user experience in a standard web browser. This architecture has become popular in the recent years in many web-enabled applications. While the web browser interface has been standard in enterprise applications for years, single-page applications are not yet very common.

2.1 Traditional web applications

Traditional browser applications process user input, then send changes back to the server as HTTP request. The server then processes the request and sends the next page to user's browser. This architecture poses two problems for software usability standpoint. First, the user interface must be re-rendered entirely when user requests another page. Secondly, the user cannot interact with application in any way while a page request is in progress.

With traditional web applications, JavaScript is typically used for decorative functions and basic data validation (Ullman 2012, 7). The web server is responsible the application logic and user interface HTML generation. Figure 1 illustrates the architecture of a traditional web application.

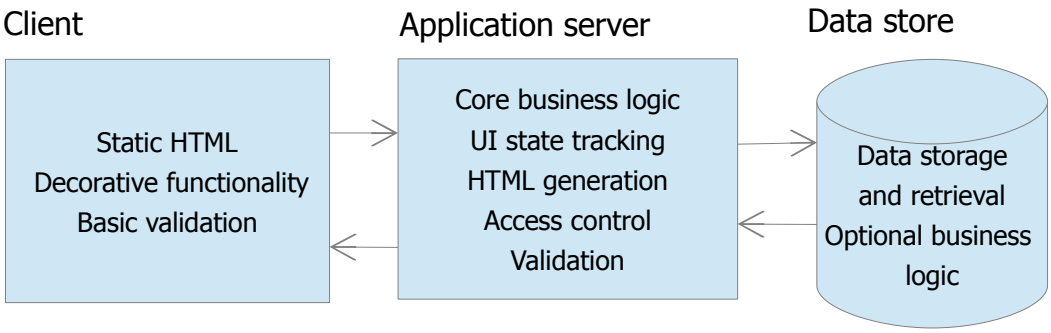

Figure 1. Architecture of a traditional web application.

2.2 Single-page web applications

Single-page applications improve the traditional web application architecture by loading the initial page just once, and transferring only application data afterwards. With properly engineered application, the user interface can stay responsive while new data is transferred from the server. Another advantage is that client side can keep track of application state without assistance of the web server. Figure 2 illustrates the single-page web application architecture.

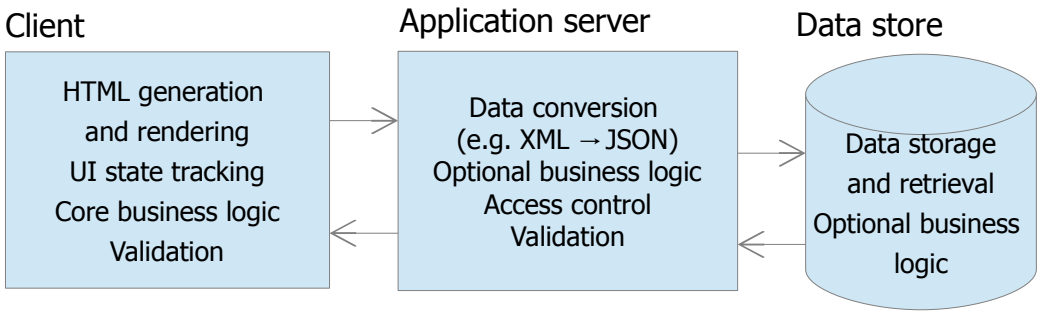

Figure 2. Architecture of a single-page web application.

Single-page Applications do not inherently depend on URLs for different views within an application. After single-page application is started, it no longer relies on the web server for changing the views or application state. Because of the independent nature of a single-page application it is also easier to support offline functionality, where relevant data is stored is in the browser local storage and then synchronized with the server when connection is again available.

As single-page application model does not follow the original browser application architecture, it also leads to some challenges that require different solutions than in traditional web applications. While traditional application errors could be inspected in server-side logs, the single-page applications do not have such option. A script error can halt the whole application, and it is then unable to connect to the server to report the error. This issue can be mitigated by testing the application in wide variety of platforms and browsers, and also by writing code to recover from error conditions where possible.

Another important issue in single-page applications is memory management. Because traditional browser applications use page transitions that clear the JavaScript context before loading each new page, memory leaks are often a non-issue because the context exist relatively little time. In single-page applications the JavaScript context is persistent over the whole application run-time life-cycle. Although JavaScript provides automatic memory allocation and garbage collection, it is important that the developer understands the allocation concepts used by the underlying framework and makes the effort to clear the references to objects that are no longer in use yet still cannot be automatically deallocated by the garbage collector.

While single-page applications move the vast majority of application logic from the server to the browser, there are still areas where data processing should be performed on the server to avoid data security issues. For example, a password hashing function could be implemented within a single-page application, but it would be bad practice from the security standpoint because the functionality could be easily reverse-engineered by inspecting the client source code.

2.3 Building blocks of single-page web applications

The JavaScript programming language is a key technology in single-page web applications. It is the only programming language that is supported on all standards-compliant web browsers. While JavaScript was originally conceived as a scripting language, it has developed to become a robust general-purpose programming language (Flanagan 2010, 1).

JavaScript is object-oriented, weakly typed interpreted language (Ullman 2012, 4). It borrows its basic syntax from Java, but it is not a member of the object oriented C family languages. JavaScript, unlike the majority of other object oriented languages, features a prototype-based inheritance mechanism. This means that instead of inheriting from a class, JavaScript objects inherit directly from other objects (Crockford 2008, 46). Another distinctive feature of JavaScript is that functions are first-class objects. This means that functions can be passed as parameters to other

functions, returned from a function, and can contain their own properties and methods (Stefanov 2010, 57).

Document Object Model or DOM defines how web applications interact with the HTML of the rendered pages (Odell 2009, 58). It can be considered as the glue between JavaScript code and the HTML user interface. It an is important technology for single-page applications, because they have to be able to dynamically build the user interface instead of relying on readily constructed HTML pages from a web server.

Single-page programming model is based around concept of separating the application data from browser DOM. This is realized by combining together DOM manipulation to dynamically build the user interface and asynchronous data requests to fetch the application data while keeping the UI responsive.

Another key technology for enabling single-page programming model is Ajax. The concept is to send and receive data between a browser and a web server asynchronously without actually navigating to another page. Ajax functionality is utilized though the XML HTTP Request object (Ullman 2012, 426). Asynchronous operation ensures that the user interface stays responsive while an operation is in progress. When asynchronous operation completes it can update the user interface partially instead of loading a whole new page.

2.4 Choice of platforms

The primary platform for single-page page web application is the standard web browser in the desktop/laptop computers. Mobile-friendly user experience can often be built in the same application using display components and style sheets that scale to smaller screens of these devices.

Hybrid applications are native applications that run inside a web browser. Therefore the web browser must be either embedded in the executable, or provided by the

operating system. Hybrid applications emerge from the need to reduce costs when building applications for multiple platforms. The advantage that hybrid application offers is native-like application user experience that is accomplished with web application developer skill set. The mobile operating system vendors typically offer access to native services of the device using specific JavaScript library to interface with this functionality.

The third option is to build a desktop hybrid application. Similarly to mobile hybrid, the main motivation is to provide native-like user experience. Applications where desktop hybrid is feasible could include software with large offline storage requirements or use of a platform service that is otherwise unavailable in web application.

3 Accessing enterprise back-end systems

Enterprise applications are by definition not self-contained. They depend on a number of different back-end systems that provide the data that is required for the application. Enterprise back-end services could consist of databases, web services, ERPs or legacy systems.

3.1 XML

Extensible Markup Language is a popular open standard for information interchange. It is designed to be both human-readable and machine-readable. XML supports unicode encoding, making it useful for transmitting documents in different languages (Moseley 2007, 168). A document consists of a hierarchy of tags that describe the names of the elements. Content is placed between start-tag and end-tag. Tags may have optional attributes to define additional metadata about the element.

XML may be utilized without any strict rules about the message structure or it may conform to a schema, a set of rules that has been defined about the structure of the included data. Both methods have their advantages. When schema is not utilized,

message format can change rapidly. With schema in use, the validity of the document structure can be determined programmatically. Validating an XML document with a schema is significantly more complex operation than just parsing the content of the document, and is not available in all frameworks and libraries that process XML.

While early JavaScript-enabled web applications commonly used XML format for transferring application data between client and the server, it is rarely used in single-page web applications. As Flanagan (2011, 493) notes, the Ajax techniques work with XML , however the use of XML is purely optional and has actually become rare.

The weakness of the format is that the tag data around the actual payload values often comprise a significant amount of the final message size. This may become an issue especially when using application over mobile network. There are also performance considerations with the use of XML formatted data. Odell (2009, 170) argues that XML parsing through JavaScript is a slow operation, and should be avoided when possible.

3.2 SOAP

SOAP is a commonly used message framework in enterprise web services. It defines a standard for exchanging XML messages over a network (Moseley 2007, 212). SOAP defines standard message format called envelope, which consists of header, body and optional fault sections (Bean 2010, 46).

Web Service Definition Language or WSDL is typically utilized to describe the web service. It is an XML document containing the service name and location, supported operations and their input and output data types. XML Schema metadata is utilized to describe the data types available.

As SOAP utilizes HTTP as a transport medium, it is accessible from a browser context. However, because it uses XML message format, the same performance considerations apply to it as with plain XML messages.

3.3 JSON

JavaScript Object Notation is a lightweight data interchange format for storing unordered name/value pairs (Crockford 2008, 136). It is a text format, allowing both human and machine readability. Although the format is a subset of JavaScript language, tools for processing it have been implemented in many popular programming languages. JSON has become a de facto standard data format in web applications (Freeman 2014, 116).

As JSON is technically a piece of JavaScript code, it can be parsed from the text representation to a JavaScript object with eval() function. Using eval() is not recommended however, because no verification of correctness is applied to parsing. The recommended parsing method is utilizing the JSON.parse() method provided by modern JavaScript environments to guard against malformed or maliciously built JSON data (Crockford 2008, 139).

While JSON data format solves many problems as an efficient data interchange format, it is not without its faults. JSON lacks a native definition for storing dates. By default the JavaScript date objects are encoded in JSON as strings. This means there is a risk of decoding errors when data is utilized across different programming languages and back-end systems which may expect different kind of encoding.

3.4 REST

Representational State Transfer or REST is a software architecture style to build web services. The requested URL identifies the data that is being operated on and the HTTP method identifies the operation to be performed (Freeman 2014, 551). REST includes no metadata to describe the service or message data content. The architecture utilizes existing HTTP command verbs to map common persistent storage activities as presented in Table 1.

Table 1. Mapping HTTP commands to CRUD activities

HTTP Command	**CRUD Activity**
POST	Create
GET	Read
PUT	Update
DELETE	Delete

Restful services are inherently stateless. This goes hand-in-hand with single-page application architecture by reducing the overall complexity of a system because it is no longer necessary to coordinate application state in the server (Richardson et al. 2007, 323).

Stateless architecture presents a challenge in transaction management when such functionality is required. There is no concept of grouping a set of REST service calls to form a transactional unit of work. Often the best way to overcome this limitation is to move parts of the business logic to server side, and executing it via web services.

3.5 Databases and legacy systems

Relational databases and legacy systems often utilize a proprietary binary protocol to access their services. At the lowest level, web browsers can only support HTTP protocol. Therefore direct access to services that require a specific protocol built on TCP/IP is beyond capabilities of a web browser. To overcome this problem, an application server or an integration platform could be utilized to convert the data to a preferred format on-the-fly.

4 Modularity and dependency management

In the JavaScript context, a module is a function or object that has a public interface but keeps its state and implementation private (Crockford 2008, 40). In addition to

promoting loose coupling, re-usability and maintainability, particularly important function of modules in JavaScript is to prevent conflicts with the global variables.

Unlike many other programming languages, the JavaScript provides no language constructs for modularizing the source code of an application (Flanagan 2011, 246). This has led to various developments to combine other language features to provide module systems in JavaScript applications.

Asynchronous Module Definition or AMD specifies a JavaScript API to define modules and to load them asynchronously (De 2014, 145). It is a popular solution for solving the problem of the lack of native module definition and loading features in JavaScript and is widely supported in libraries and frameworks.

5 Build process and application deployment

At first it may seem unnecessary to have a build process for an interpreted language, since there is no compilation or linking process required to produce a runnable application. However, there are many areas where project data needs transforming in order to become production ready.

Minification is a process of removing information from source code that is unnecessary from the point of code execution at run-time (Stefanov 2010, 36). This means removing white space and comments while renaming variables and functions to have as short as possible names. Minification process significantly reduces application size and improves run-time performance when loading pages.

Another performance optimization is combining different source code files together in one file. This process is referenced as concatenation (Foster et al. 2014, 100). Concatenation allows to reduce the amount of HTTP calls to fetch the necessary JavaScript files and thus optimize the application start-up time.

A common build process task is to perform a static source code analysis that finds code that is syntactically valid but potentially a cause of program errors. This process is referred as linting. The lint tool has its roots in early C language compilers, where it was used to supplement the compiler warnings. A popular tool for performing this analysis for JavaScript code is JSLint. It defines a stricter subset of the JavaScript that is better suited to professional development (Crockford 2008, 115).

6 Data binding and validation

6.1 Templates

Template systems convert data into display elements (Mikowski et al. 2014, 209). They are used to separate the application logic from the presentation. A template provides a skeleton HTML fragment with placeholders for dynamic data. When the page is rendered, the framework applies the dynamic data to the templates.

There are two styles of templating systems: embedded and toolkit. Toolkit style template systems have no standardized syntax, thus each templating engine defines its own ways of building the user interface using a domain-specific language. Embedded style templating systems utilize the host programming language to build the template representation. Mikowski et al. (2014, 209) recommend against using an embedded style templating systems, because they make too easy to intermingle business logic with display logic.

There are multiple options on where the template data can be defined. The only restriction is that the framework's templating engine must be able to access them at application run-time. Popular options are embedding templates in the application view code, embedding inside HTML script tags or using separate files for template data.

6.2 Binding data to views

Application frameworks based on the Model-View-Controller design pattern hold the application data entities in model objects (Pressman 2005, 580). When the application displays data to user and when user makes changes to that data, the application must have a way to synchronize changes between the model and the user interface. Data binding refers to the process of linking data from the model with what is displayed in a web page (Dayley 2014, 400).

One-way binding means that a value is taken from a model and inserted into a HTML element (Freeman 2014, 237). It is suitable for data that is not modified by the user. Two-way binding means that changes are tracked in both directions (Freeman 2014, 240). This allows both changes in the model to be reflected in the user interface, and changes made by the user to be reflected in the data model.

Some application frameworks provide automatic two-way data binding. This allows developers to build views without needing to manually write synchronization code to keep track of data model and user interface changes. As enterprise applications often consist of many form based views with a significant amount of user input, this is an important consideration when choosing a framework. With manual data binding, a lot of developer work could go into extracting user input from the different views and moving the data between the DOM and the model.

6.3 Validating user input

Validation is the process of ensuring technical correctness of the data entered by the user. For example, a phone number typed in by the user might be required to conform to a specific set of rules in order to be considered valid. Validation was one of the first feasible uses for JavaScript in early web applications (Ullman 2012, 10).

There are multiple ways to implement validation. The first option is to program the validation rules using string manipulation and other language constructs of the JavaScript programming language. However, JavaScript also supports standard regular

expressions. Regular expressions can be used to write compact, rule-based templates (Moseley 2007, 144). Template patterns are tested against user input to determine the validity of the data. Using regular expressions allows more compact representation of validation rules, so it is preferred over programmatic checking of the input data.

The natural advantage of a single-page application is that the data validation can be much more extensive, because application model data resides fully in the client. This way the validation process can use full model information to implement the validation rules that can have complex dependencies instead of basing the rules on data only shown on the current application view.

7 Testing

Testing is an important part of software quality assurance. Automated testing can support both the application building process and the validation process. Software defects can be categorized in two classes. A new defect is an error that appears after introducing a new feature in the software. A regression defect is an error that reappears in previously working part of the software. Saleh (2003, 9) argues that the number of new defects and the regression defects becomes unmanageable when the code base becomes complicated and unit testing is not available.

7.1 Unit testing

Unit testing focuses verification effort on a software component or module (Pressman 2005, 394). Passing conditions of a unit test is defined by assertions. The assertion function validates that a condition is valid, and if not, it throws an error. If a unit test throws an error, the test is considered to have failed. A test method can include one or more assertions. All assertions have to pass in order for the test method to pass (Saleh 2003, 74).

Pressman (2005, 395) argues that selective testing of execution paths is essential task during the unit test. Test case design should take into account possible errors in computation, incorrect comparisons and improper flow control.

Mock objects implement the API of the required components but generate predictable pre-generated results. The behavior of the mock objects is altered to create different scenarios in which to test code (Freeman 2014, 632). This allows testing the component in contained environment without actually needing to reconfigure the application environment.

7.2 Integration testing

Pressman (2005, 397) defines integration testing as a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing. Already unit tested components are connected together incrementally to ensure that they work correctly together and that their interfacing is not producing unintended side-effects.

Compared to unit testing, largely the same testing tools can be utilized to design and run integration tests. However, because different components and their interfaces have to be set up independently, the effort required to build an integration test could be significantly larger than building a simple unit test.

7.3 Continuous integration

When development team members develop software independently, integrating changes to common code base may result in unforeseen consequences. Continuous integration is a process that helps to mitigate this problem. Roemer (2013, 138) defines the role of continuous integration to aggregate and test application code to detect integration errors early and automatically.

Continuous integration is typically implemented in a dedicated server that constantly monitors changes in the code base. When a change is detected, application source code is checked out from the revision control system. Then the server proceeds to build the application and run its test suite. Finally, any build errors or test failures are reported to the development team.

7.4 Testing considerations in single-page applications

JavaScript, being a dynamic language offers some advantages in automated testing compared to traditional statically typed languages. Mock objects of the business data structures are easily dynamically constructed at run-time, and they can be stored in JSON format without any external library dependencies.

Application code should be written and structured with a mindset that allows isolation of different components of the software. This in turn allows easier implementation of unit tests. According to De (2014, 136) writing tests consistently will make code more structured, flexible and easier to use by other team members.

Because single-page applications construct their views by dynamically altering the DOM, it is guaranteed that at least some parts of the application depend on having the DOM objects available in the test environment. This can be accomplished by running tests in a browser simulator. A browser simulator library provides a JavaScript API for manipulating the DOM objects similarly to a real web browser (Roemer 2013, 140). The advantage of a browser simulator is that the test suite can be run in command-line scope without any dependencies to web browsers installed in the system.

Testing with actual web browsers requires careful consideration. The application requirements could demand compatibility with a broad range of different hardware and software platforms that could make the effort to run test cases for all combinations of configurations considerable. On the other hand, if the application targets a hybrid platform, the browser is provided either by the platform or

embedded in the application package narrowing the range of testing targets considerably. Mikowski et al. (2014, 143) argue that if browser interaction is required, the value of automated testing is diminished greatly as the cost of implementation rises.

8 Framework study: Backbone.js

Backbone.js is a JavaScript application framework for building web applications. It aims to provide a minimal set of tools to structure a web application (De 2014, 8). The framework is open sourced and distributed under the MIT license. It can be easily integrated with existing web applications because the different library components can be utilized on case-by-case basis (Foster et al. 2014, 10). This allows developers to upgrade existing applications gradually.

Backbone.js provides the basic building blocks for constructing applications: Model, Collection, View, Router, Events and Sync. Application data entities are represented as Models. Models of similar type are accessed as Collections. For user interface, Backbone.js provides Views to display application data, and Router to connect the Views to form a full application. Events are utilized to send messages between objects. Finally, Sync connects Models and Collections to a web service back-end for data persistence.

Because of the minimal design approach, different extensions extending specific parts of Backbone.js have become popular. Many extensions are available and they are useful for supplementing the core Backbone.js feature set based on development project requirements. Minimal design approach also enables developers to quickly study the source code and perform changes on fundamental framework to suit project needs.

Backbone.js depends on jQuery and Underscore.js libraries. The jQuery library provides Backbone.js a unified access to DOM, event handling and support for REST API (Foster et al. 2014, 10). Underscore is a generic utility library with an emphasis on

functional style programming. It aims to supplement JavaScript's low number of built-in utility methods and to enable writing of more intuitive and concise code (De 2014, 10).

8.1 Views

View encapsulates any logical part of the user interface. It manages event handling, model changes and business logic in a specific region within the application. Views enable developers to build re-usable components and enhance the modularity of the application (Foster et al. 2014, 41).

The concept of a view should be applied without any prejudice. A valid use for a Backbone.js view could range from a single data item in a list all the way to a landing page of an enterprise application. Different views serving different roles can have varying life-cycle: a landing page of an application will persist for entire lifetime of an application session, while data item in a list sub-view could be very short-lived.

If a view has references to external objects, they should be cleared when the view is destroyed to allow JavaScript engine garbage collection to work. If a method remove() is defined in view, the framework will automatically call it on removal of the view. Failure to clear object references leads to memory leaks (Foster et al. 2014, 50).

Backbone.js provides basic templating support provided by the Underscore library. Templates can consist of HTML snippets and JavaScript statements to build the UI representation of the model data. Template data can be embedded in the DOM or defined in the view as a JavaScript string.

Views can be extended to provide hierarchy of generic and specific views. When building views that have similarities, common functionality can be moved to a base class. This way the boilerplate code required to build each view can be reduced.

8.2 Models

In Backbone.js, the Model object provides the basic data object. As enterprise applications are often backed by relational database, the model objects typically represents one row of data from a specific table in a database. At first it may seem confusing why the framework provides a model type which could be represented with standard JavaScript object, however after studying the model object it becomes obvious that it provides much more features that a standard JavaScript object.

The first problem with standard JavaScript objects is that it is a non-trivial problem to detect if the object state is changed (Foster et al. 2014, 56). Backbone.js models work around this by providing get() and set() methods that keep track of attribute changes.

Models have built-in for support framework events that allow views to react to model changes without additional functionality. This means that a view can listen to a particular event of a model, and when the event is triggered, the view can update its content automatically to reflect the change in the model.

Models have support for unique id property. When models are refreshed from the back-end database, models with same ids are automatically merged. If a model has urlRoot property, the framework can refresh its data from the back-end with the fetch() method. The save() method allows to persist the model data to back-end. A model data can be deleted from the back-end using destroy() method.

8.3 Validation

In Backbone.js the validation is linked to the model objects. A model can perform basic validation on its own data. This is accomplished by overriding the validate() method. Validation is performed automatically when the model is saved on the back-end. Validation can also be performed on model changes when validate parameter supplied to set() method options.

A custom validate() method should return true if the model passes its validation requirements. Any other return value is considered a validation failure (Foster et al. 2014, 62). The return value can be used to provide an object with key/value pairs containing the fields that have failed the validation, and the error messages that describe why the validation failed. This is useful for displaying validation errors in the user interface.

8.4 Events

Event handling in Backbone.js consists of two aspects. DOM event selectors and their event handler functions are defined in the view's events property as key-value pairs. These events are scoped to the part of the DOM that the view is responsible to render.

Another part of the Backbone.js event handling is object events. A view can respond to object events (e.g. a model or collection) and automatically re-render the view when the underlying data model changes. The following Backbone.js objects provide the events API built-in: Model, Collection, View, and Router along with the Backbone.js main object (Foster et al. 2014, 31). In addition, events can be mixed in with any object to provide for custom event handling needs.

Events are sent using a trigger() method with any Events API enabled object. The trigger expects event name and optional arguments as parameters to be passed to event listeners. The event names can be chosen freely, and there is no need to pre-register events. The downside of this is that developer must take care when naming the events to avoid name clashes between different components of the application.

Backbone.js main object can be used as a global event bus and utilized as a global publisher/subscriber event service across a whole application. Because the Events object can be mixed with any object, events can be utilized in more modular scope. This is again one example where Backbone.js presents flexible programming model, forcing neither global nor fine-grained object level events.

When the underlying view is destroyed, Backbone.js automatically removes event listeners that are set up using listenTo() or listenToOnce() methods (Foster et al 2014, 50). However, developer must manually remove events that are initialized using on() method and events that are bound to external objects to avoid memory leaks in the application.

8.5 Collections

Collection could be viewed as a result of a database query. It is a set of models that share a similar type. Collections maintain their ordering and they can be sorted in any way necessary by providing comparator function to reorder the data. Collection can accept both models and standard JavaScript objects. Standard objects are automatically converted to models before they are added in to a collection.

If a comparator function is specified, it is invoked whenever a new model is inserted (Foster et al. 2014, 70). Collections automatically keep the set of included models duplicate free. Models that already exist in the collection are merged with the existing collection entries.

To integrate collections with persistence, similar interface is provided as with the Backbone.js models. If an URL parameter is provided to a collection, its contents can be loaded from the back-end with the fetch() method, and stored with the save() method. To re-synchronize the collection data from the back-end server, collections provide the update() method.

Collections support the Backbone's event framework. This way views can listen for specific changes when the user interface needs to change according to changes in the collections instead of needing to manually poll application data for changes.

When working with web services provided by a third party, it is often necessary to make some conversions at the incoming data. For example, the raw data from a web

service may contain headers or metadata in addition to what is expected to be inserted into a collection. For these cases, Backbone.js collections provide parse() method, which can be utilized to convert the data to a more desirable format.

8.6 Persistence

Sync object provides the means to load and store persistent data to server back-end. When models and collections are loaded and saved, the sync object does most of the actual work even when the developer might never call it directly.

By default Backbone.js provides persistence implementation with REST support utilizing JSON data format web services. Overriding the default implementation could provide any kind of protocol customized for the back end. For example, a XML based protocol could be implemented to access a custom enterprise back-end.

The sync object can be re-purposed to support multiple data sources. This way an application could utilize normal enterprise back-end when the user is online, but revert to browser local storage for data persistence when offline.

8.7 Routing and History

In single-page applications, the browser does not load a URL from the server on application state transitions. Instead the views are created and destroyed by the application logic. The router object maps persistent URLs to different views of an application and tracks the application state. Variables can be encoded in parameters of the URL.

Relative URLs in the application and their respective routing actions are defined as key-value pairs in the routes property. The routing action receives any parameters supplied in the URL. It is the responsibility of the routing action to initialize a view and signal it to display a specific application state.

Another related aspect related to routing is the history object that provides browser navigation history support. This is necessary because technically single-page application stays on one page for the whole life-cycle of the application. The history object enables forward and backward buttons in the browser to work similarly to traditional web applications.

8.8 Module system

Backbone.js provides no module system by its own. However, it supports the common Asynchronous Module Definition format for defining and loading modules. If no module system is used, each JavaScript source file must be loaded in its own HTML script tag (De 2014, 110). This approach has two problems. First, it will be necessary to define the script tags in correct order to avoid errors related to a script referencing another script that is not yet loaded. Secondly, the browser must load each source file sequentially resulting poor application loading performance.

The Asynchronous Module Definition avoids these problems. Developer can specify the dependencies in any order, and the module system automatically loads them in correct order. Loading also happens asynchronously which enables parallel module loading for better performance.

9 Framework study: AngularJS

AngularJS is a web application framework that is sponsored and maintained by Google (Freeman 2014, 3). It is distributed under open source MIT license. The framework enforces a Model-View-Controller application structure. AngularJS brings many familiar server-side concepts to single-page application context, like dependency injection and concept of services.

AngularJS is self-contained in terms of library dependencies. If AngularJS detects that jQuery is available at start-up, it will be used. If no jQuery is available, AngularJS uses

a light-weight internal implementation called jQlite. This implementation provides a subset of jQuery library features that enables AngularJS to run (Freeman 2014, 400).

9.1 Application

AngularJS application is implemented by placing special tags in the HTML view template. These special HTML elements and attributes instruct the framework to interact with AngularJS components that are used to build an application.

AngularJS components have different responsibilities, however, they share a number of common qualities. Components that have dependencies on each other support dependency injection (Freeman 2014, 212).

Application life-cycle consists of three phases: bootstrap, compilation, and runtime data binding (Dayley 2014, 401). Bootstrap occurs when the initial application page is loaded and the browser is instructed to load the AngularJS JavaScript library. In the bootstrap phase the framework is initialized and the application root directive is located.

The second step in the application life-cycle is the compilation phase. It occurs after bootstrap is completed, and presents the initial view using static DOM. In the background the framework traverses the static DOM and proceeds to collect and link the directives with their respective scopes to provide the dynamic view.

The final step in the life-cycle is the runtime phase. The applications stays in runtime phase until the page is reloaded, when the user navigates away from the page, or when the browser is closed. In runtime phase the framework constantly updates the view to reflect the changes in scope.

As AngularJS compiles the templates in the second application life-cycle phase, it trades faster runtime performance of views for slower application start-up time. As Freeman (2014, 45) notes, the AngularJS application start-up time is often negligible

with modern browsers in desktop computers, however, this might become an issue with mobile browsers running on less powerful hardware.

9.2 Controller

Controllers are objects that manage the data flow between models and views. A typical AngularJS application consists of multiple controllers for each aspect of the application. AngularJS conventions recommend controller names to be suffixed with Ctrl (Freeman 2014, 212).

AngularJS uses $scope object to interface the data binding between controller and view. When a new instance of a controller is created, a new child scope that is specific to that controller is created as well (Dayley 2014, 420). The controller is responsible for the business logic attached to that scope. A controller is applied to a view using the ng-controller directive.

9.3 Views and Templates

AngularJS utilises HTML templates that contain special directives to instruct the framework to do the page rendering. The main page should be considered as layout template, and sub-views should be used to modularize the implementation of different aspects of the user interface.

A view that is used as a sub-view to another view is applied as a partial template. When the framework encounters an ng-include directive, it automatically retrieves the HTML fragment and adds it to the Document Object Model (Freeman 2014, 250). No boilerplate HTML is required in the partial template as the main view provides the base layout around the fragment.

AngularJS template contains standard HTML combined with AngularJS specific elements and attributes to build the view that is presented to the user. HTML portion

provides the static parts of the user interface, while expressions, filters and directives manipulate the DOM to present the dynamic data from the scope model.

Expressions

Expressions are code that is evaluated within a scope. They are commonly used to produce a dynamic value to HTML component text or attribute. Expressions are enclosed in double brackets.

Expressions behave in many ways like JavaScript code, however, there are important differences in scope and flow control. Property names evaluate against scope model instead of global JavaScript namespace. No flow control conditionals, loops or throwing errors within expressions are allowed.

Filters

Filters are a key element in AngularJS when model data needs to be transformed from original representation to more user-friendly format. AngularJS provides many standard filters for manipulating models and collections. Example filters for models include formatting strings, dates or numbers. For collections, standard filters include a sorting or limiting number shown of entries in a list.

Multiple filters can be chained together, and it is also possible to create custom filters. Custom filters are created using Module.filter() method by providing a factory function to generate a worker function (Freeman 2014, 351). The role of the worker function is to perform the actual data transformation or formatting.

Directives

Directives are special HTML elements or attributes that are specific to Angular. They enhance and modify these elements to provide application functionality, such as event handling or data validation. Like expressions, developer can utilize scope model variables and use expression syntax when defining directives.

Developer can also implement custom directives when built-in directives are not applicable. Custom directives are built using Module.directive() method. A directive can navigate and modify the HTML elements anywhere in the Document Object Model using jQuery selectors. Freeman (2014, 401) recommends to only modify the children and descendants of the element passed to the directive to ensure that other directive operations are not interfered.

9.4 Scopes and Models

AngularJS uses standard JavaScript objects to represent application data models. Models are exposed to the view by appending them to the $scope object in a controller (Dayley 2014, 399). Scope object manages the data used in views, business logic and server back-end.

AngularJS provides two-way data binding. This means that the model state is automatically synchronized between the controller and the view. Model properties are dirty checked by the framework in order to ensure that changes stay in sync. In cases where the model is changed outside of the AngularJS context, the developer can manually instruct AngularJS to update the bindings using the $apply() method.

A hierarchy of scopes can be created by nesting the controller directives in the template. Child scopes are able to access parent scopes from a controller, however, not vice versa (Dayley 2014, 426). If an already existing property is added in a child scope, the property in the parent scope is not overwritten, but a new property is created in the child scope.

Scopes can send and receive application events. $emit() method in scope sends an event through parent scope hierarchy. All scopes that are ancestors to that scope can receive the event. Similarly, $broadcast() method in scope is used to send an event thought the child scope hierarchy. Each event has a mandatory name and optional extra arguments that the event receiver can use as needed.

Scopes receive application events by setting up an event listener using $on() method. This method expects to receive an event name and handler function as arguments. As event names are global, care should be taken to avoid name clashes among different events across scope hierarchies.

In addition to view specific scopes, AngularJS provides application level root scope. The $rootScope object can be injected to other components and utilized as a global data store for the application.

9.5 Validation

In AngularJS, the validation is performed by a combination of HTML element attributes and AngularJS specific directives in the HTML template. The framework honors the standard HTML element attribute keywords such as type and required (Freeman 2014, 294). When the standard elements are not flexible enough to validate the user input, the validation directives can be utilized.

The framework provides several built-in directives that are suitable for validation. The ng-minlength and ng-maxlength directives allow to specify minimum or maximum length of the user input. The ng-pattern uses a standard regular expression pattern that must match the contents of the validated element for validation to succeed. The ng-change directive evaluates an expression against the input element, and the expression returning true indicates successful validation. If none of the provided directives are suitable for the validation task at hand, a custom directive can be written to handle these situations.

AngularJS makes use of HTML classes to report the validation results. Valid element receives the ng-valid class, while on invalid element the ng-invalid class is inserted. The classes can be combined with Cascading Style Sheets to report the validation status in the user interface. The framework updates these elements after every interaction allowing instant feedback on user actions (Freeman 2014, 300).

9.6 Services

Services are singleton objects meant to handle business logic. To promote loose coupling and re-usability, view-independent logic should be implemented in a service instead of a controller.

Life-cycle of a service differs from a controller. Controller life-cycle is tied to a view, and when user switches to another view, the controller is destroyed. Services on the other hand are instantiated at application start-up, and they survive the lifetime of the application.

There are both built-in and custom services. A common example of a built-in service is $http, that is used to make AJAX request to a server. Custom services built by the developer behave exactly as built-in services. They support all capabilities of AngularJS components, such as dependency injection.

Custom services can be tied specifically to one application component and might have no other uses. On the other hand, a service that addresses a broad scope of requirements could be reused in multiple applications.

9.7 Dependency Injection

Dependency injection or DI means resolving component dependencies and instantiating them automatically at runtime (Freeman 2014, 212). It allows easier integration between different modules in application and reduces initialization code. While dependency injection is widely used in server-side programming languages, it rarely implemented in JavaScript (Dayley 2014, 411).

Because JavaScript is a dynamic language, it presents additional challenges for dependency injection. Static data type declarations do not exist in source code and thus cannot be used as hints to determine what dependencies should be injected to a

variable. AngularJS works around this limitation by looking at the parameter names in a function and then proceeds to inject dependencies with matching names available at the injector service.

AngularJS provides dependency injection through providers and the $injector service. Provider functionality is defined by developer in AngularJS components. The $injector service can then provide dependency injection to automatically resolve these dependencies at run-time (Freeman 2014, 616).

An important consideration with dependency injection is that it should be verified that the possible code minification in the application build process is not breaking the application functionality. This is most likely to happen if implicit annotation syntax is utilized in Angular. In most cases, the inline annotation syntax is the most robust way to implement dependency injection in Angular, and does not suffer any side effects from the use of code minification tools.

9.8 Routing and History

Routing in AngularJS is handled by the $route service. It is responsible for mapping application URLs to partial templates. New routes are added by calling the when() method and specifying route and template information in the method parameters.

Routes consist of static path components and route parameters. Parameters can match one or more segments depending on their configuration. Two types of route parameters are available: conservative and eager. The conservative parameter will match only one segment, while the eager parameter tries to match as many segments as possible (Freeman 2014, 590). Conservative parameters are returned to the route handler as-is. Eager parameters are automatically grouped into an array. They are particularly useful for handling optional values in the application routes.

9.9 Module system

The framework provides integrated support for modularizing an application. Each view has a single module assigned to it via the ng-app directive (Dayley 2014, 399). Additional modules can be added to main module as dependencies.

An AngularJS module is built by defining the objects that it provides. Full application is built by linking different modules through dependency injection. In large scale applications it is important to divide the components into logical modules. Building one directional dependencies between modules promotes loose coupling and leads to more maintainable and reusable code base.

A module consists of a configuration phase and a run phase (Dayley 2014, 412). The configuration phase executes when a module is defined. Any providers are then registered with the injector service. Run phase executes after configuration phase, and implements necessary operations to instantiate the module.

There are no technical limitations on how modules can be organized. Different components of the same logical module can be arranged in different source files. Popular strategies are organizing by feature, where all components are placed in a module that implements one application feature, or organizing by type, where all components of same type are put into a module.

9.10 Testing

AngularJS provides an optional module called ngMock, which provides tools for unit testing (Freeman 2014, 624). The module needs to be downloaded separately because it is not intended to be included in production release of an application.

The ngMock module helps to isolate the different components in the software by facilitating the creation of mock objects of the components that are required by the component under test. It provides mock services of commonly used built-in services, such as $http or $log. The angular.mock object provides methods that load modules

and allows dependencies to be resolved in the unit tests (Freeman 2014, 632). Combining these features together the ngMock module enables all aspects of component inputs and outputs to be simulated and measured, allowing a broad coverage of unit tests to be written.

10 Conclusions

Single-page browser applications provide many technical advantages compared to traditional web applications. Modern web technologies enable user interface responsiveness and usability that closely matches desktop applications. Complex logic can be applied to the data presented in the user interface without a round-trip to the web server. Another technical advantage is that the application state is fully managed on client. Traditional web applications have separate state tracking in client and in server, which leads to duplication of responsibility and is a potential source of application errors.

Another consideration is the development team skill set aspect. With single-page application frameworks the majority of a web application can be constructed using just one programming language. In traditional web applications there is always JavaScript as client-side language, and some other programming language that is used to implement the server-side functionality. While single-page applications in enterprise context will usually still need an application server to provide the back-end services, the majority of the application development work is moved to the client and therefore there is less need for server-side expertise.

The challenges with single-page applications in enterprise context often lay in the data integration. Enterprise software commonly relies on multiple back-end systems using varying protocols to access them. Accessing these systems as-is from a single-page web application could range from easy—such as a REST web service utilizing JSON data format—to impossible—such as a relational database or legacy system utilizing a binary protocol. As enterprise application development teams typically have experience with server-side technologies, it is often the easiest route to use an

application server or integration platform to convert and proxy the various enterprise services to the single-page application in an easily digestible format.

Backbone.js is a good match when the application in development has need for highly flexible and customizable framework. Backbone.js also has an advantage when applied to an existing application as retrofit, because the framework feature set can be applied partially and the framework itself is very light-weight.

AngularJS is well suited for a typical form based applications that involve a lot of user input processing. The automatic two-way data binding enables better productivity, because developers have no need to write view and model synchronization code manually. Built-in module system and dependency injection allow building a highly decoupled and modular application structure to ensure a maintainable code base when the project grows. Finally, the integrated testing facilities help isolating the software components in a project allowing thorough unit testing.

Both frameworks are released under open source license that enables studying and modifying the frameworks to better suit the project under development. However, the minimal approach of Backbone.js makes it easier for a developer to understand the inner workings of the framework and modify or extend its functionality as necessary. Because AngularJS is larger both in scope and code base, it makes more sense only to extend its functionality at documented extension points. At the same time, when the application requirements are well aligned with the AngularJS feature set, it would take a non-trivial amount of extensions and/or modifications to Backbone.js in order to match the scope of AngularJS.

Final consideration is the learning curve. Developers with web background should find Backbone.js easy to learn, because it brings a more structured approach to otherwise already learned skill set. AngularJS takes a different route, providing a design that is built on a number of features that are more common on the server-side platforms. On the other hand, developers with enterprise application background are often familiar with concepts like dependency injection from the similar

implementations in server-side languages and can quickly take advantage of these features.

References

Bean, J. 2010. SOA and Web Services Interface Design.
Morgan Kaufmann.

Crockford, D. 2008. JavaScript: The Good Parts.
O'Reilly Media.

De, S., 2014. Backbone.js Patterns and Best Practices.
Packt Publishing.

Dayley, B. 2014. Node.js, MongoDB and AngularJS Web Development.
Addison-Wesley.

Flanagan, D. 2011. JavaScript: The Definitive Guide, Sixth Edition.
O'Reilly Media.

Foster, C., Feldman, A., Tonge, D, Freo, P., Branyen, T. 2014. Developing a Backbone.js Edge. Bleeding Edge Press.

Freeman, A. 2014. Pro AngularJS.
Apress Media.

Gillham, B. 2010. Case Study Research Methods.
Continuum International Publishing.

Moseley, R. 2007. Developing Web Applications.
John Wiley & Sons.

Odell, D. 2009. Pro JavaScript RIA Techniques.
Apress Media.

Mikowski, M., Powell, J. 2014. Single Page Web Applications.
Manning.

Pressman, R. 2005. Software Engineering: A Practitioner's Approach. Sixth Edition.
McGraw-Hill.

Richardson, L., Ruby, S. 2007. RESTful Web Services.
O'Reilly Media.

Roemer, R. 2013, Backbone.js Testing.
Packt Publishing.

Saleh, H. 2013, JavaScript Unit Testing.
Packt Publishing.

Stefanov, S. 2010. JavaScript Patterns.
O'Reilly Media.

Ullman, L. 2012. Modern JavaScript: Develop and Design
Peachpit Press.

Appendices

Appendix 1: Common application concepts between Backbone.js and AngularJS

Concept	Backbone.js	AngularJS
Application event handling	Events can be sent using trigger() method and handled using on() or listenTo() methods.	A scope can send events upward in hierarchy with $emit() method and downward with $broadcast() method. Events are handled using $on() method.
Library dependencies	Underscore.js and jQuery	No external dependencies, although jQuery will be used if found
Modularizing view independent business logic and utility methods	AMD modules should be used to separate logic from view code.	Utility methods and business logic are placed in services.
Routing	Backbone.Router object defines routes hash where route strings and handlers are matched.	$routeProvider.when() method is used to map URL routes, templates and controllers.

Appendix 2: Common MVC concepts between Backbone.js and AngularJS

Concept	Backbone.js	AngularJS
Collection data	Backbone.Collection represents a set of Backbone.Models.	Collections are standard JavaScript arrays.
Collection ordering	A comparator function can be specified for a collection. A collection is reordered according to comparator when new entries are added.	Standard JavaScript constructs for ordering arrays are used
Data binding	Developer must write view specific code to update the model and the DOM as needed.	Model and DOM changes are automatically kept synchronized.
Data persistence	Models and collections are loaded and stored to back-end using fetch() and save() methods.	Developer must write controller specific code for loading and storing changes on the back-end.
DOM event handling	Each view has events object where source selectors and target handler functions are defined.	Events are bound using directives in templates.
Model data	Each view object has model property that is used to assign model data specific to the view. Models are implemented by extending Backbone.Model.	Each view has $scope object where model data is appended. Models are standard JavaScript objects.
Validation	Model.validate() method is executed by the framework before a model is saved to the back-end. Validation should return either true or an object with failed field names and error descriptions as key/value pairs.	Validation directives in the input element attributes are configured in the template (e.g. ng-change, ng-minlength, ng-maxlength, ng-pattern, ng-required)