

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

OLEKSANDR IEFIMENKO
BACHELOR THESIS

A TOOL FOR VISUALIZING PATCH-FLOW

Submitted on May 1st 2016

Supervisors: Prof. Dr. Dirk Riehle, M.B.A., M.Sc. Maximilian Capraro
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander University Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, May 1st 2016

License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see <https://creativecommons.org/licenses/by/4.0/>

Erlangen, May 1st 2016

Abstract

Inner source is the use of open source software development practices and the establishment of an open source-like culture within organization that helps to improve code reuse and share knowledge where product line engineering fails (Capraro et al. 2016). We measure the inner source collaboration by measuring the code contributions (*Patch-Flow*) between project boundaries or organizational units.

The management of software development organizations, which want to adopt the inner source strategy in Enterprise, needs a tool for visualizing *Patch-Flow*, helping it to analyze the collaborative process and make the software development process within companies more effective. Nowadays, the market cannot offer any product to supply the demand.

This thesis develops a software design and implementation of tool to represent the various *Patch-Flow*-based metrics for quantifying code-level collaboration and stake of participants in it. The presented tool is the first product allowing quantitative visualization of *Patch-Flow*. It enables managers to evaluate and make decisions about the code-level collaboration and supports them in the inner source context.

Content

Introduction.....	6
1 Initial Data and Objectives.....	7
1.1 Organizational Model	7
1.2 Visualizations.....	7
1.2.1 Hierarchical Organizational Summary	7
1.2.2 Force-Directed Patch-Flow Graph.....	8
1.2.3 OrgUnit- / Project-related Patch-Flow Chart.....	8
1.2.4 Contribution Ledger.....	9
2 Client-Server Architecture	11
2.1 REST-Architecture.....	11
2.2 Single Page Application.....	11
3 Evaluation of OSS components	13
3.1 Java-based RESTful Framework	13
3.2 JavaScript-based Framework for SPA	16
3.3 Routing with Angular UI-Router	17
3.4 Angular Smart Table	17
3.5 Angular Chart Frameworks.....	17
4 Software Design.....	19
4.1 REST-API	19
4.1.1 Resources Identification.....	19
4.1.2 Representation Format and Media Type	21
4.1.3 Designing URIs.....	22
4.1.4 Versioning	22
4.2 Processing a Client Request: Service Layer	23
4.3 Export Utility	23
4.4 Static Factory Method.....	24
4.5 Angular Routing.....	24
5 Implementation	26
5.1 Server with Jersey Framework.....	26
5.1.1 Setup with Maven Archetype.....	26
5.1.2 Example of Resource Implementation.....	26
5.2 Server Testing	28
5.2.1 Configuration.....	28
5.2.2 Unit Testing with Mocks.....	29
5.3 Angular Client.....	29
5.3.1 Application Setup.....	29

5.3.2	Directory Layout.....	30
5.3.3	MVC Pattern	30
5.3.4	Routing.....	30
5.3.5	REST Client	32
5.3.6	Force-directed Patch-Flow Graph.....	33
5.3.7	Testing.....	34
	Conclusions and Future Work.....	35
	References.....	36

Introduction

The idea of inner source to apply the best *Open Source* (OS) methods for the internal development process was first mentioned around the late nineties, soon after OS attracted more attention from researchers and the software industry (Argefalk et al. 2015). Adopting of OS practices within an organization brings a lot of advantages such as a code reuse across the organization, without complex discussions at a management level and a corporate bureaucracy, more effective testing with fewer bugs during deployment, improved quality of documentation. Riehle et al. (2015) defined inner source as an approach to collaboration across intra-organizational boundaries for the creation of shared reusable assets.

The code-level collaboration in inner source takes place by contributing patches. Patch is an incremental contribution to a software project. Software developers are able to send patches (a flow) with bug fixes, new features or other additions, which can be later accepted or rejected by owners of inner source components (Capraro et al. 2016). *Patch-Flow* is the flow of patches contributed across intra-organizational boundaries such as project or organizational unit.

To enable a measurement-driven inner source management, the collaboration needs to be measured. Research Group developed a *Patch-Flow crawler*, which gathers the real *Patch-Flow* data from various repositories. Now we need a tool to visualize the *Patch-Flow* and *Patch-Flow*-based metrics on this basis, which enables management to be kept informed how much collaboration is happening, who are the contributors and receivers, what is the stake of an organizational unit in the collaboration.

In this thesis, I based on an implemented model for measuring *Patch-Flow* data with a prototype of database filled with contribution records, and developed a client-side web application for its representing. The contributions of this paper are following:

- Definition of diagrams to visualize *Patch-Flow*
- Evaluation of *Open Source Software* (OSS) components for implementing the diagrams
- Development of a client-server application implementing defined diagrams

This paper is structured as follows. Chapter 1 describes a background of application (the *Patch-Flow* process) and defines the types of developing visualizations. Chapter 2 considers the client-server architecture. In Chapter 3 I evaluate and choose OSS components that are going to be used in the web application. Chapter 4 provides a software design. Chapter 5 focuses on an implementation of developing tool.

1 Initial Data and Objectives

The goal of this thesis is to enable the easy management of collaboration by providing several visualizations of *Patch-Flow*. The preexisting *SCM Crawler* tool gathers the *Patch-Flow* data from numerous remote repositories and stores it into a database based on PostgreSQL.

1.1 Organizational Model

In the scope of this thesis I use a preexisting organizational model. It helps to identify patches and match them to contributors and receivers. There are few steps to determine the *Patch-Flow* data within an organization:

1. Identify patches
2. Identify **receiving projects**
3. Identify **contributing projects**
4. Match projects to **hosting organizational units**

I illustrated the organizational model with references to the *Patch* class in the Figure 1.1 and marked steps and appropriate references with different colors for the correct matching.

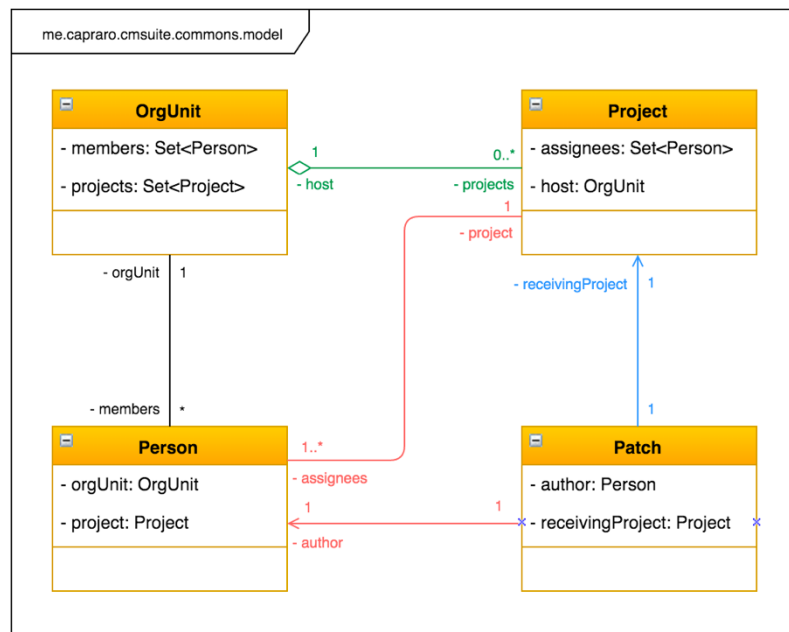


Figure 1.1: Organizational model with patch references

1.2 Visualizations

1.2.1 Hierarchical Organizational Summary

An organizational chart in Figure 1.2 provides a clear visual representation of the organization's units as a tree. It shows a hierarchical structure of organization in a form "parent-child" and informs users about how many patches were contributed and received by any organizational unit. The balance value is displayed and colored in green (more patches received), in blue (more patches contributed) or in gray (a difference is equal to zero).

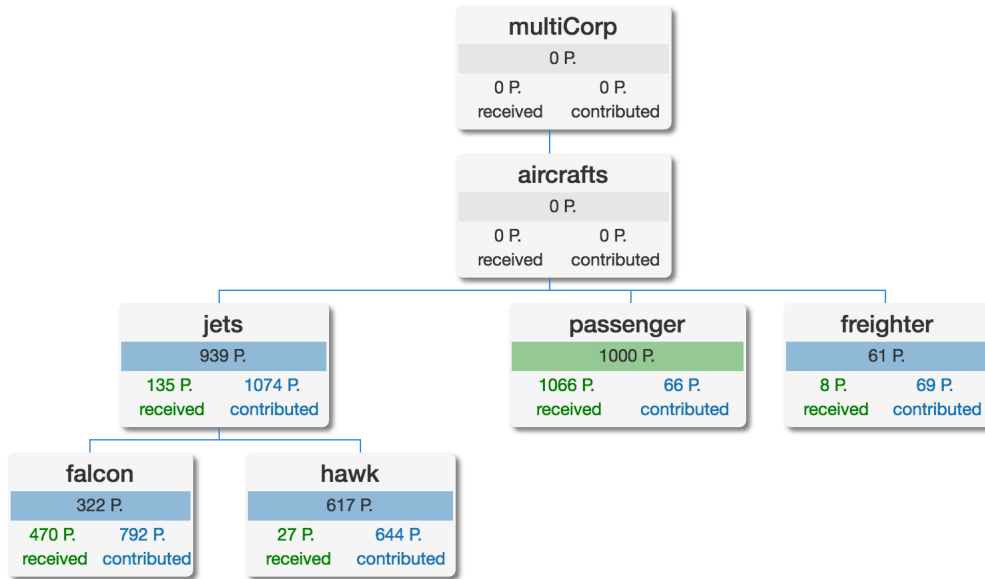


Figure 1.2: Example of an organizational chart

1.2.2 Force-Directed Patch-Flow Graph

One of the most significant features which should be provided by developing tool is a force bi-directional *Patch-Flow* graph (Figure 1.3). It supplies the users with a total statistics of *Patch-Flow* within an organization. There are two graphs in total: one graph displays the *Path-Flow* between organizational units, another one illustrates the flow between projects. Each graph consists of nodes and bi-directional links, where the nodes map organizational units or projects and links represent the *Patch-Flow* between them. It is possible for users to move or rotate the graph as much as they would like to do.

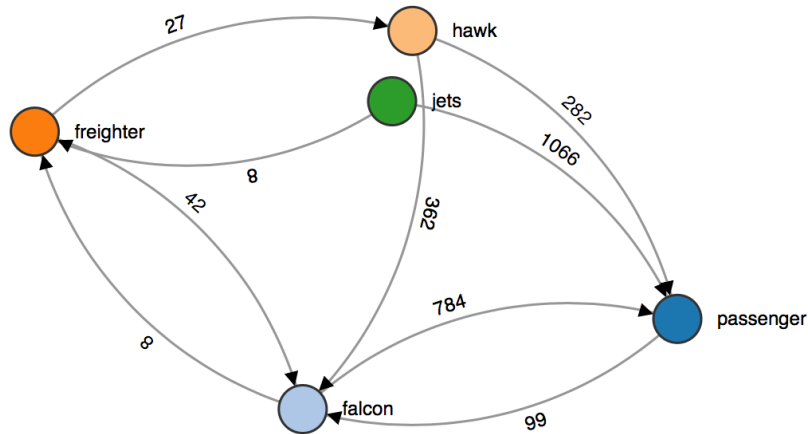


Figure 1.3: Example of a bi-directional *Patch-Flow* graph

1.2.3 OrgUnit- / Project-related Patch-Flow Chart

By visualization of code-level collaboration, I distinguish between *Patch-Flow* for incoming patches (received) and *Patch-Flow* for outgoing patches (contributed) per project or organizational unit in total. All charts display metrics over a selected time range. All performed contributions (commits) are displayed in gray, received patches in green and contributed patches in blue. Example of these charts is shown below.

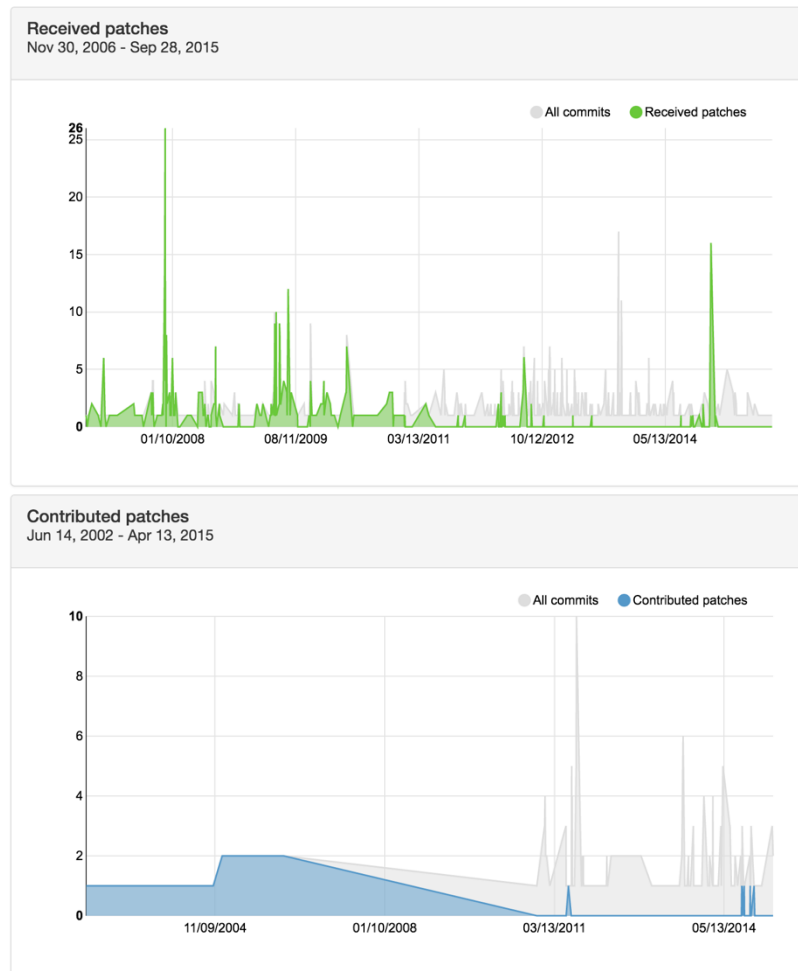


Figure 1.4: Example of Patch-Flow charts

Normalization

The tool also provides a normalization for both metrics. The example diagrams are shown below.

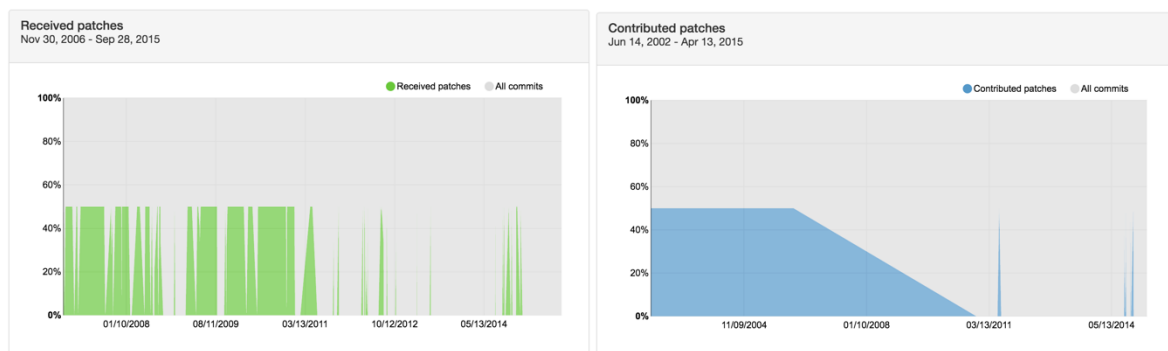


Figure 1.5: Patch-Flow normalization

1.2.4 Contribution Ledger

The contribution ledger shows all patches received and contributed by project or organizational unit in a list like an overview. There are two views of a horizontal bar chart: a total summary (Figure 1.6) and a grouped summary by project token (Figure 1.7). Each view displays the

account's summary consisting of an amount of received and contributed patches and their difference, the balance. There are different ways to sort or filter the listed data: by date, by amount or by name.

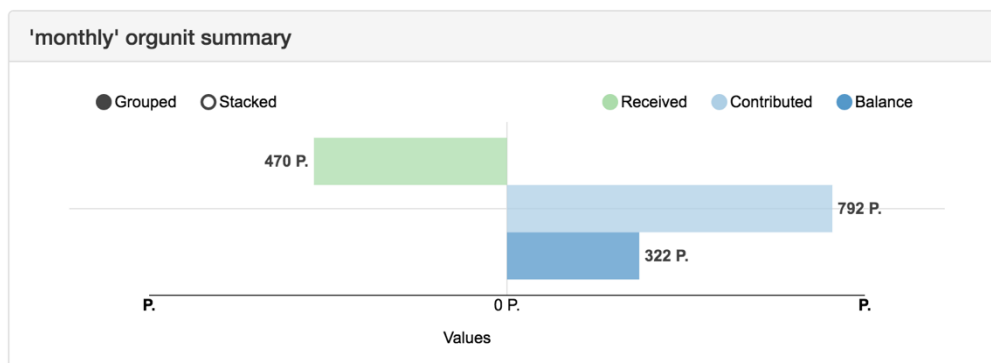


Figure 1.6: Example of a contribution ledger (total)

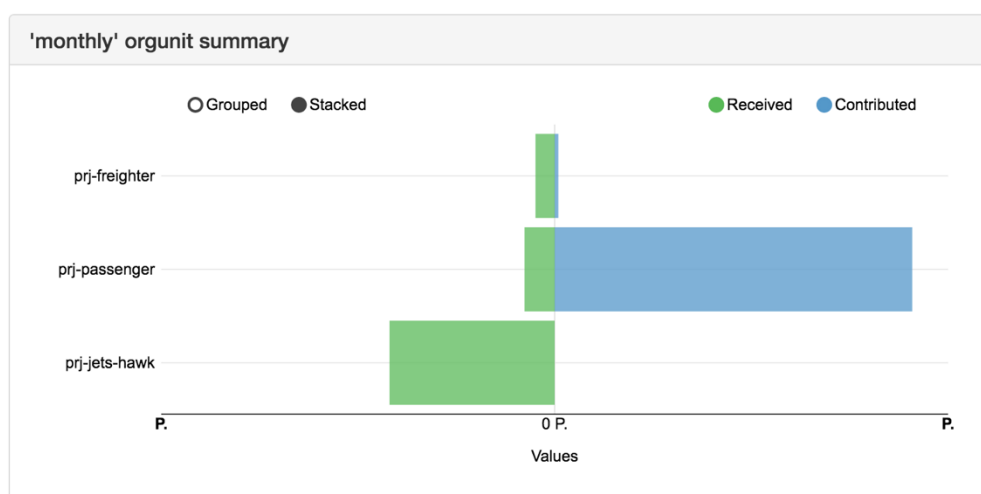


Figure 1.7: Example of a contribution ledger (grouped by project)

2 Client-Server Architecture

Delivering concepts, methods and tools to enable a measurement-driven inner source management, sets the following questions:

- How much collaboration take place?
- Who are contributors and receivers?
- Is everything good or some actions needed?

To make it possible to answer these questions, I developed a tool called *CMSuite Explorer* based on a client-server model. This distributed application structure consists of a client requesting resources and a provider of services and resources, a server. The client-server model describes a relationship between these two participants in the web application. Clients and servers communicate over HTTP protocol. A server program, for sharing their resources with clients and for formalization of data exchange, must implement a web service. A web service enables communication by applying one of the open standards such as HTML, XML, SOAP, etc. HTTP, the web server and web services work together to serve data by clients requests. Implementations of web services provide web APIs. One of them is a RESTful API, which doesn't require XML-based web service protocols such as SOAP and WDSL to access its interface. REST offers a simple, interoperable and flexible way of writing web services.

2.1 REST-Architecture

A web server is a central part of a client-server architecture. It processes clients requests performed over HTTP protocol. For effective interaction between components over a network and data exchanging I need to implement a web service.

REST or *Representational State Transfer* architectural style for distributed hypermedia systems is firstly introduced by Roy Fielding in his PhD thesis (Fielding, 2000). He defined a set of architectural constraints such as uniform interface, stateless, cache, etc., which bring a performance and a scalability, and enable services to work best on the Web (Oracle Documentation, 2010). However, the first specified constraints come from the client-server architectural model. The client-server style is the most frequently encountered of architectural styles for network-based applications (Fielding, 2000). In the following Chapters I am going to evaluate and choose an OS framework implementing RESTful API and describe steps in its developing for my application.

2.2 Single Page Application

The goal of this Section is to clarify an architecture of client. We should distinguish between traditional multipage web applications and *Single Page Applications* (SPA). In a traditional web application, when a client requests a server's content or service function, the server renders a new HTML page which caused a page refresh in the browser. Figure 2.1 illustrates the second approach. SPA loads a single HTML page and updates the page regions with new fragments dynamically through AJAX calls without constant page reloads. It means that much of the work takes place on the client side using JavaScript. In the past few years SPA has become extremely popular on the web providing a more fluid user experience with UI. In the following Chapter I make a review of most popular JavaScript-based frameworks to make SPA development more easily.

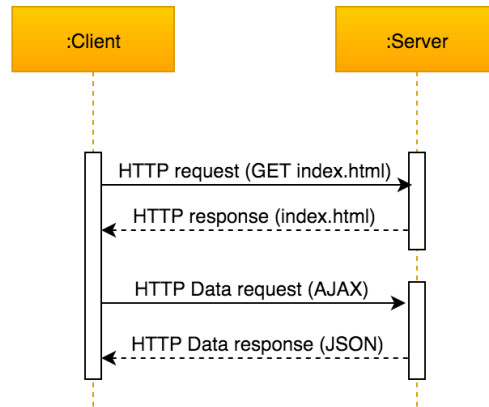


Figure 2.1: Lifecycle of simple Single Page Application

3 Evaluation of OSS components

In this Chapter, I present the comparative analysis of OSS frameworks and libraries for development of the client-server application. I used the following criteria for selecting OSS:

- review of gathered pros and cons
- activity of OS community (Open Hub, GitHub)
- result of a benchmark test (for RESTful frameworks)
- presence of a functionality (concerning chart libraries)

3.1 Java-based RESTful Framework

A table below lists OS Java-based frameworks, which participate in this evaluation. They help with a development of RESTful web services.

Name	Specification / Routing	License
Dropwizard	Jersey	Apache 2.0
Jersey	JAX-RS + Extending API	CDDL 1.0, GPL 2
Ninja	Own routing with one central route file, JAX-RS (optional)	Apache 2.0
Play	Own approach with routes file	Apache 2.0
RESTEasy	JAX-RS	Apache 2.0
Restlet	JAX-RS	Apache 2.0, CDDL 1.0
Spark	Own route patterns	Apache 2.0

Table 3.1: Java-based frameworks

For gathering of pros and cons I used a public data and documentation from official web sites and several articles such as (Gaic, 2015), (Gupta, 2014), (Spark, 2015).

Dropwizard

Dropwizard provides stable, mature Java libraries assembled into a simple, light-weight package providing us with everything that lets us focus on getting things done. It is fast and modular with a large community support. This framework uses the modern Java based web application components available, packaged into an easy-to-use framework but this also makes it problematic. Mixing all those libraries can cause unforeseen problems in the large projects. Another problem is the maintenance: earlier versions are hard to update.

Jersey

Jersey provides its API that extends existing JAX-RS reference implementation with additional features and utilities to further simplify RESTful service and client development. It is fast with extremely easy routing and smooth JUnit integration for testing. Jersey supports true asynchronous connections, has an excellent documentation with working examples and has one of the best IDE-support, so I can achieve better productivity from tooling perspective. This OS framework has a reference implementation for JAX-RS so it will continue to support any updates to the specification. The significant advantage with a large amount of online resources is the disadvantage at the same time. It is related to Jersey 1.X making them unsuitable for Jersey 2.X.

Ninja

Ninja is a full-stack web framework for Java. Stable, fast and super productive. The benefits also include a modularity, multiple rendering formats, a supporting of other libraries and caching. Comparing with Jersey, Ninja has a small community and a provided documentation is also not good enough, same like with Dropwizard.

Play

Play is based on a lightweight, stateless, web-friendly architecture and makes it easy to create, build and deploy web applications with Java or Scala. This framework is built on Netty so it supports non-blocking I/O, excellent for RESTful applications with handling remote calls in parallel. It also supports a modularity, MVC and has probably the largest community among frameworks reviewed here. Some of the drawbacks are that Play 2.x is not compatible with Play 1.x, it is not based on JAX-RS and hard to understand and configure. There are breaking changes across releases.

RESTEasy

RestEasy may be a good choice if the environment is JBoss oriented. It provides a proprietary caching for URL or query which could be handy for high volume applications, and asynchronous HTTP abstractions. The disadvantage of this framework is a small community.

RESTlet

This is a first RESTful web framework for Java, implemented REST before it was popular. JAX-RS was a natural extension. It offers a smart URL binding, a powerful routing and a filtering system and is available for all major platforms but it is complex and has a closed community. RESTlet is not as popular as these days comparing with Play framework and Jersey.

Spark

Spark is a micro lightweight Java web framework made for a rapid development with minimal effort. It has a minimalist core providing with all the essential features. This framework is the most commonly used with AngularJS. In its own survey from April 2015 (Spark, 2015) responded by a couple of hundred users, most of them answered that's Spark documentation is not good enough. Amongst other known disadvantages Spark is not suitable for large projects and has a small community, so this framework is mainly good for smaller projects and fast prototyping.

One of the important factors by choosing a framework is a community. The comparison result provided with Open Hub (Black Duck Open Hub, 2016) is listed in the following table. As we may see, there are three active communities among these projects: Play with 855 active developers and Dropwizard followed by Jersey.

Framework	All time statistics		12 months statistics	
	Contributors	Commits	Contributors	Commits
Dropwizard	248	6,568	98	1,078
Jersey	116	6,327	29	410
Ninja	76	1,785	26	279
Play	855	11,431	175	1,672
RESTEasy	130	4,396	38	386
Restlet	52	8,523	6	28
Spark	81	655	32	279

Table 3.2: The comparison result of communities' activity on Open Hub for Java-frameworks (15.04.2016)

By evaluation I also used a benchmark test of web frameworks performed by *TechEmpower* on February, 25th 2016 (TechEmpower, 2016). The test was performed as follows:

- The framework's ORM fetched all rows from a database table containing an unknown number of Unix fortune cookie messages.
- An additional message was inserted into the list at runtime and then the list was sorted by the message text.
- The list was delivered to the client using a server-side HTML template.

In this benchmark test (Table 3.3) Jersey with 52,801 points achieved the best result among others participants of my review.

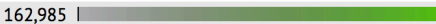

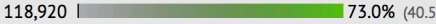





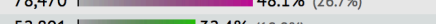
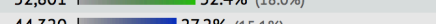
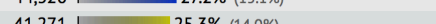
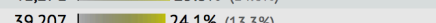
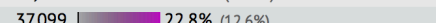
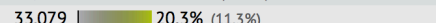
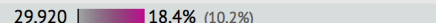
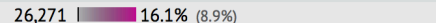
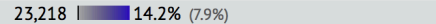
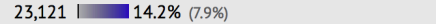
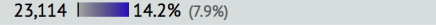
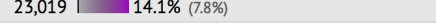
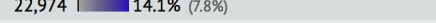
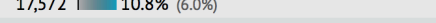

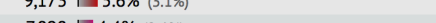
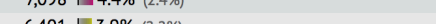
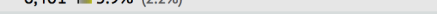
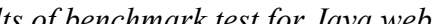
Framework	Best performance (higher is better)	Cls	Lng	Plt	FE	Aos	DB	Dos	Orm	IA	Errors
geminii-postgres	162,985 	Ful	Jav	Svt	Res	Lin	Pg	Lin	Mcr	Rea	0
servlet-raw	119,813 	Plt	Jav	Svt	Res	Lin	My	Lin	Raw	Rea	0
geminii-mysql	118,920 	Ful	Jav	Svt	Res	Lin	My	Lin	Mcr	Rea	0
undertow	105,273 	Plt	Jav	Utw	Non	Lin	My	Lin	Raw	Rea	0
undertow edge	103,088 	Plt	Jav	Und	Non	Lin	My	Lin	Raw	Rea	0
undertow	91,595 	Plt	Jav	Utw	Non	Lin	Pg	Lin	Raw	Rea	0
undertow edge	91,069 	Plt	Jav	Und	Non	Lin	Pg	Lin	Raw	Rea	0
undertow	80,360 	Plt	Jav	Utw	Non	Lin	Mo	Lin	Raw	Rea	0
undertow edge	78,470 	Plt	Jav	Und	Non	Lin	Mo	Lin	Raw	Rea	0
grizzly-jersey	52,801 	Mcr	Jav	Svt	Grz	Lin	My	Lin	Ful	Rea	0
play-java-jpa	44,320 	Ful	Jav	Nty	Non	Lin	My	Lin	Ful	Rea	0
dropwizard	41,271 	Ful	Jav	Jty	Jty	Lin	Pg	Lin	Ful	Rea	0
dropwizard	39,207 	Ful	Jav	Jty	Jty	Lin	My	Lin	Ful	Rea	0
tapestry	37,099 	Ful	Jav	Svt	Res	Lin	My	Lin	Ful	Rea	0
wildfly-ee7	33,079 	Plt	Jav	Svt	Wil	Lin	My	Lin	Ful	Rea	0
undertow-jersey-hika	29,920 	Mcr	Jav	Svt	Utw	Lin	My	Lin	Ful	Rea	0
undertow-jersey-c3p0	26,271 	Mcr	Jav	Svt	Utw	Lin	My	Lin	Ful	Rea	0
sabina jetty mongodb	23,218 	Mcr	Jav	Svt	Non	Lin	Mo	Lin	Raw	Rea	0
sabina	23,121 	Mcr	Jav	Svt	Non	Lin	Mo	Lin	Raw	Rea	0
sabina jetty mysql	23,114 	Mcr	Jav	Svt	Non	Lin	My	Lin	Raw	Rea	0
spring	23,019 	Ful	Jav	Svt	Tom	Lin	My	Lin	Ful	Rea	0
sabina undertow mysq	22,974 	Mcr	Jav	Svt	Non	Lin	My	Lin	Raw	Rea	0
ninja-standalone	17,572 	Ful	Jav	Jty	Jty	Lin	My	Lin	Ful	Rea	0
vertx-web-mongodb	10,533 	Mcr	Jav	vtx	Non	Lin	Mo	Lin	Raw	Rea	0
activeweb	9,173 	Ful	Jav	Svt	Res	Lin	My	Lin	Mcr	Rea	0
vertx-web-jdbc	7,098 	Mcr	Jav	vtx	Non	Lin	Pg	Lin	Raw	Rea	0
dropwizard-mongodb	6,401 	Ful	Jav	Jty	Jty	Lin	Mo	Lin	Ful	Rea	0

Table 3.3: Results of benchmark test for Java web frameworks

Abbreviations	Definiton
Cls (Ful, Plt, Mcr)	Classification (Fullstack, Platform, Micro)
Lng (Jav)	Language (Java)
Plt (Jty, Nty, Svt, Und, vtx)	Platform (Jetty, Netty, Servlet, Undertow Edge, vertx)
FE (Grz, Jty, Non, Res, Tom, Utw, Wil)	Front-end-sever / Web server (Grizzly, Jetty, None, Resin, Tomcat, Undertow, Wildfly)
Aos (Lin)	Application operating system (Linux)
DB (Mo, My, Pg)	Database (MongoDB, MySQL, Postgres)
Dos (Lin)	Database operating system (Linux)
Orm (Ful, Mcr, Raw)	ORM Classification (Full, Micro, Raw)
IA (Rea)	Implementation approach (Realistic)

Table 3.4: Abbreviations and their definition used in the benchmark test

In summary, Jersey is one of the most popular frameworks in the REST community with a significant performance among other frameworks. The performance is one of the most important aspects that characterizes a success of a new product. Jersey framework can it guarantee.

3.2 JavaScript-based Framework for SPA

There are different OS JavaScript frameworks which make it easier to create a powerful single web application. I chose the most popular of them listed in Table 3.5.

Name	Example Applications	License
AngularJS	Freelancer, Netflix, Tinder, Google Analytics etc.	MIT License
Backbone.JS	Foursquare, Disqus, Airbnb	MIT License
Ember	Yahoo!, Groupon	MIT License

Table 3.5: JavaScript-based frameworks

All the listed frameworks have a lot in common: they are open-sourced and implement SPAs using the MVC design pattern. In this Section, I am going to compare these three frameworks basing on the Uri Shaked's article (Shaked, 2014).

AngularJS

First of the comparing UI-frameworks is an AngularJS, a JavaScript-based MVC framework for building rich client-side applications. Its benefits include two-way data binding, saving a lot of boilerplate code, a versatility and flexibility enabling the software developers to achieve a result by several ways, a dependency injection, easy-to-test-code. The framework provides ready-to-use, powerful mocks for fundamental built-in services like *\$http*. Generally, AngularJS is perfect for smaller applications or components. Its disadvantage is the complexity of the directives API.

AngularJS provides the automatic “dirty checking” and automatically detects all changes after modifying any property of a scope object, and notify all the watchers for that property. But this feature, for monitoring the changes, may also cause problems with updating an interface when there are a large number of bindings.

Backbone.JS

Backbone.JS is a lightweight and fast MVC framework with a simple code and great documentation. It has one of the largest communities of users. Another advantage is its stability and a linear learning curve. The drawback of Backbone.JS is that it does not provide a structure and developers have to use some basic tools to create it. Also, it could take a time to find a good solution for some other features which Backbone.JS does not provide itself. The framework does not support a two-way data binding comparing with other two members of my survey, meaning developers have to write a lot of boilerplate code. Backbone directly manipulates the DOM with views, making them really hard to unit test and less reusable.

Ember.js

The solution is also based on the MVC pattern. Ember has a steep learning curve what requires from developers to follow a convention so it is more suitable for large-scale, more ambitious and long-lived projects. Ember's API has many breaking changes, a lot of the code examples might be found online are out of date. In spite of listed disadvantages, there are some benefits too. A major goal in design with Ember is a good performance. Ember.js is also helpful by configuration and unlike the two other frameworks provides a powerful data module for developing and testing.

The comparison result provided with Open Hub (Black Duck Open Hub, 2016) is listed in the following table.

Framework	All time statistics		12 months statistics	
	Contributors	Commits	Contributors	Commits
AngularJS	1,765	11,784	469	4,047
Backbone.js	330	3,306	46	378
Ember.js	698	12,181	207	2,718

Table 3.6: The comparison result of community activity on Open Hub for Java-frameworks (15.04.2016)

A result in the table shows that AngularJS has the biggest amount of active developers among comparing OS frameworks. As of March 2016, AngularJS is also being the 4th most starred project of all time on *GitHub* (GitHub, 2016). The number of questions on *StackOverflow* is bigger than by other competitors Backbone.JS and Ember together. It increases a development time and makes a maintenance process easier. Relying on the results of this survey I chose AngularJS framework for my client.

3.3 Routing with Angular UI-Router

Numerous developers do not use a native AngularJS' router because of lack of advanced routing. Angular *UI-Router* module provides an alternative solution to *ngRoute* module with supporting multiple views and nested states. With *ngRoute* only one view is allowed per page, with *UI-Router* there are multiple views and each can have own controller. This can be very useful for large projects. Nested states are used, for example, for a scenario with a list/detail page, when we need to view the details with a master list at the same time. With *ui-sref* directives this routing framework allows to change URL for any state in one place. With *\$stateParams* it is possible to pass any information between states even if it is not a part of state's URL.

3.4 Angular Smart Table

In the contribution ledger a table lists all received and contributed patches by project or organizational unit. There are several table modules for AngularJS to easily display data with filtering, sorting or other built features in a declarative way. The most popular OS developments are *UI Grid* and *ngTable* directives but I chose the lightweight *Smart Table* module with less than 4Kb. It has no other dependencies, is robust, modular and extensible. *Smart Table* can work with asynchronous data loading and provides all the features I need.

3.5 Angular Chart Frameworks

OS offers different libraries and frameworks to create the common Angular charts. For my app I chose two OS chart libraries: *Angular NVD3* and *Angular Google Chart*. For the visualization of a contribution ledger I use an AngularJS directive for NVD3 re-usable charting library based on D3. It provides the most popular charts and allows easily to customize all the charts through JSON API.

Comparing to the other OS AngularJS friendly chart frameworks, Google API provides a powerful organization chart. Org chart is a diagram of a hierarchy of nodes, commonly used to portray superior/subordinate relationships in an organization (Google Developers, 2015). The use case is to visually represent a parent-child-tree of organizational units in the nodes. Using

CSS classes this is relative easy to add the additional information containing each org unit: an amount of contributed and received patches incl. their difference value (balance) for each node.

Besides various charts, *CMSuite Explorer* provides a graph to visualize *Patch-Flow* between organizational units and projects within an organization. Mike Bostock published an example of directed graph in his blog (Bostock, 2016), which I took for the basis for my further implementation. This flexible force bi-directional graph is implemented in D3.js. A pseudo-gravity force keeps nodes centered in the visible area, while links are fixed-distance geometric constants (Bostock, 2015).

4 Software Design

A server project consists of 3 main modules (Figure 4.1): a *restservice*, a *service* and a *datahandler*. The *restservice* contains a Jersey implementation for REST-API. The inner layer *service* is a layer between *restservice* and *datahandler* to provide further operations with retrieved objects from database. And the last module *datahandler* executes queries with the database and delivers results on the *service* requests.

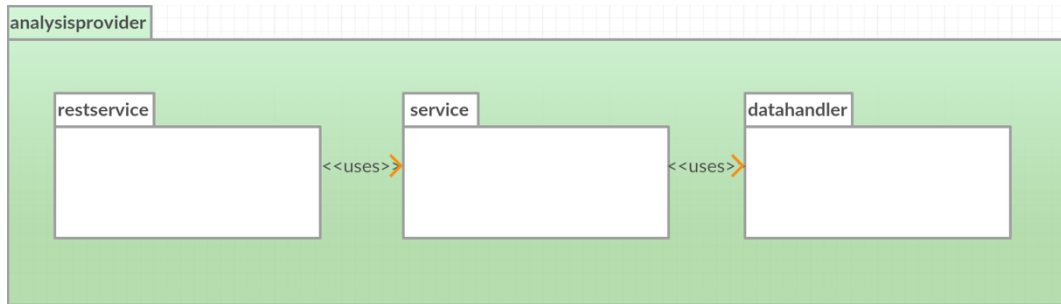


Figure 4.1: Structure of server project

4.1 REST-API

Consider the first package *restservice*, which provides a RESTful API.

4.1.1 Resources Identification

One of the first steps in developing the RESTful web service is designing the resource model. The resource model identifies and classifies all the resources the client uses to interact with the server (Allamaraju, 2010). A resource is the key abstraction of information in REST. Any information (data or functionality) can be a resource (Fielding, 2000). To uniquely identify a web resource and make it addressable the web provides *Uniform Resource Identifier* (URI). The RESTful web services are typically based on four main HTTP methods: *Create*, *Retrieve*, *Update* and *Delete* (CRUD). In my case I use only GET-method defining a reading access to the resource and getting a representation in return. It does not change the state of associated resource. In response to my GET request the server sends the status code, the response headers and the entity body as the representation in the end.

Hierarchical Organizational Tree and Patch-Flow

Considering a simple use case for the *Patch-Flow* web service: a user chooses any organizational unit or project and expects to see the appropriate *Patch-Flow* data. It is straightforward that the server should be able to deliver the list of organizational units with assigned projects and the list of all relevant patches after a client (AJAX) request.

OrgUnit class contains all information for building a hierarchical organizational tree with assigned projects. At the same time the *Patch* class contains enough data about a single patch. Both these classes (Figure 4.2) should be represented in JSON format.

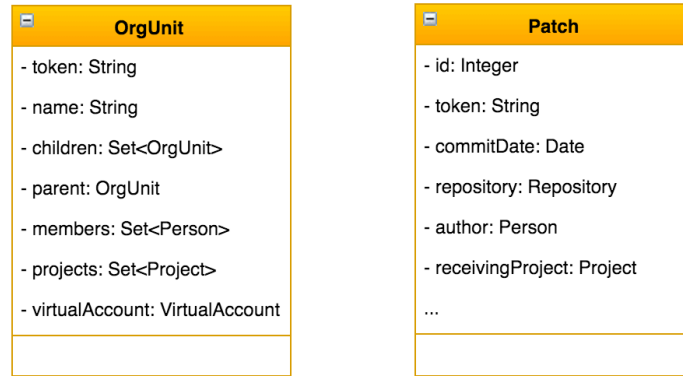


Figure 4.2: *OrgUnit and Patch classes*

A JAX-RS resource is an annotated POJO providing different methods that are able to handle HTTP requests (Jersey Documentation, 2016). Jersey framework is a reference implementation of JAX-RS. The *OrgUnit* resource can provide two resource methods: *getAll()*, a root organizational unit with all its children, and *get()* for retrieving data of a single org unit. These methods produce responses with response message content to all relevant client requests. The *Patch* resource is can be composed of only one resource method with *orgUnitToken* parameter to retrieve all the relevant contributions from database. Figure 4.3 lists the example methods of both resources.

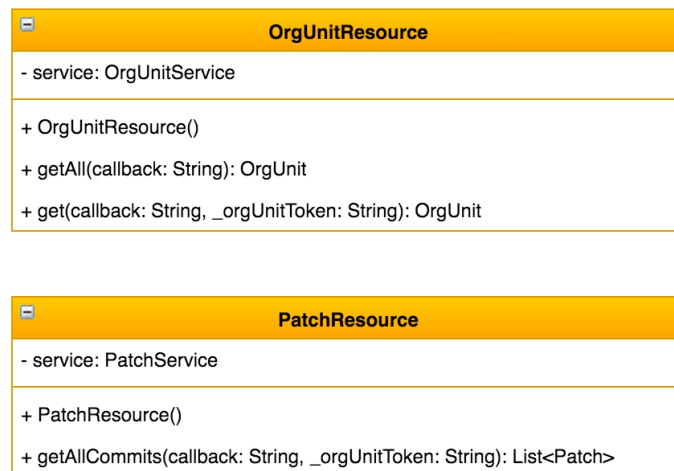


Figure 4.3: *OrgUnit and Patch resources*

Accounting

In inner source accounting model two abstractions are introduced: an *account* and a *virtual account*, where the account is an abstraction for a project and the virtual account is used for an organizational unit.

A simple use case for an Accounting page: select a project or organizational unit and show the chart with an amount of contributed, received patches and balance followed by a table with detailed information about code-level collaboration. The *Account* and *VirtualAccount* classes (Figure 4.4) contain all needed data.

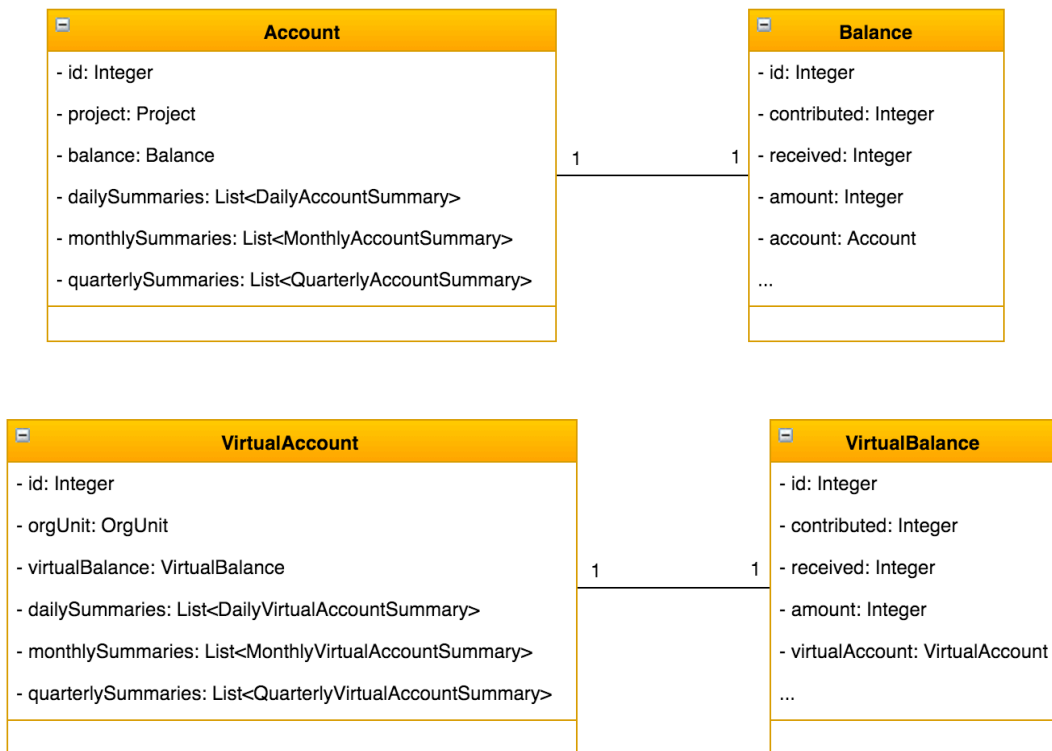


Figure 4.4: Accounting model

The collections of summaries can have an enormous size. Moreover, not all summaries we need to display on the page at the same time. *Accounting* and *VirtualAccount* resources may provide three methods for summaries, separately for each time interval. User decides which summary should be retrieved from the server.

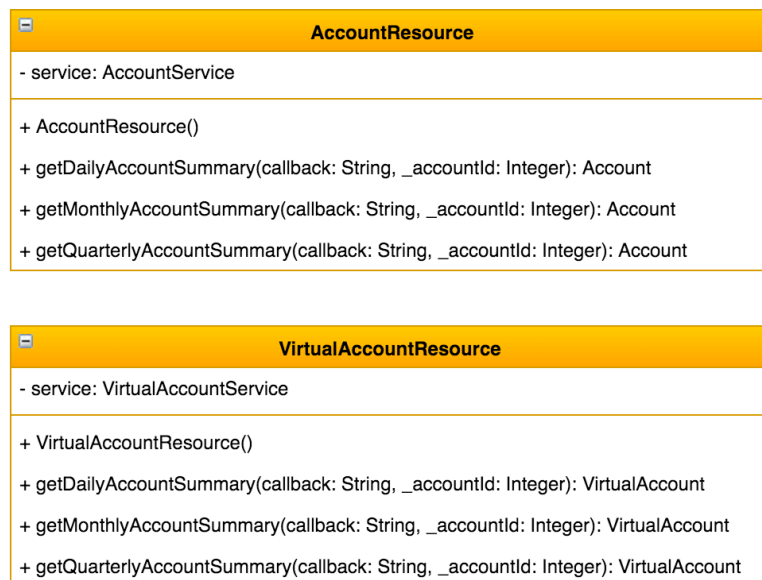


Figure 4.5: Account and VirtualAccount resources

4.1.2 Representation Format and Media Type

As mentioned above a *resource* is an abstraction or more specific an abstract entity. A representation, on the other hand, is concrete and real since that is what my program to and operates

upon in clients and servers (Allamaraju, 2010). HTTP's message format allows different media types and formats for requests and responses. A choice depends on a specific use case. Some resources may require XML-formatted representations; others need a JPG-format. I choose JSON and JavaScript as the representation formats and "application/json", "application/javascript" as Media Types. This choice matches my use case: JS format is for processing by JS-capable clients and JSON is the most popular format nowadays.

4.1.3 Designing URIs

URIs are identifiers of resources that work across the Web. Typical URI consists of a scheme (such as http or https), a host (such as www.example.com), a port number followed by a path and a query string. I design URIs following common conventions and best practices listed in (Allamaraju, 2010). For example, I use the forward-slash operator (/) in the path of the URI to indicate a hierarchical relationship between resources:

```
http://www.example.com/webapi/v0.4/orgunits/orgUnitToken
```

```
http://www.example.com/webapi/v0.4/accounts/3/monthly
```

4.1.4 Versioning

Sometimes we need to change the server code, which can break the clients. It can be also possible that some clients have a different functionality from other clients or we need to maintain separate versions of the server programs. Versioning may be a good solution when there are problems to make changes compatible for all clients. There are a variety of ways to version the REST API:

- Using detectable patterns such as v1, v2 in subdomain names, path segments, or query parameters to distinguish URIs by their version (Allamaraju, 2010)
- Custom media types in Accept header

The table below shows the result of my web research (18.11.2015) how top API providers handle versioning

API Name	Versioning	Example (source link)
Twitter	URI	https://api.twitter.com/1.1/users/show.json https://dev.twitter.com/rest/reference/get/users/show
Instagram	URI	https://api.instagram.com/v1/users/{user-id} https://instagram.com/developer/endpoints/users/#get_users
LinkedIn	URI	https://api.linkedin.com/v1/people/ https://developer.linkedin.com/docs/rest-api
Groupon	URI	<a href="http://api.groupon.com/v2/divisions?client_id=<your api key>">http://api.groupon.com/v2/divisions?client_id=<your api key> https://sites.google.com/site/grouponapiv2/api-usage
Dropbox	URI	https://api.dropboxapi.com/1/account/info https://www.dropbox.com/developers-v1/core/docs
Disqus	URI	https://disqus.com/api/3.0/users/details.json https://disqus.com/api/docs/users/details/
GitHub	MediaType	Accept: application/vnd.github.v3+json https://developer.github.com/v3/#current-version
Amazon	URI, Parameter	/myObject?versionId=3/L4kqtJlcpXroDTDmpUMLUo http://docs.aws.amazon.com/AmazonS3/latest/API/RESTObjectGET.html
YouTube	URI	https://www.googleapis.com/youtube/v3/channels https://developers.google.com/youtube/v3/sample_requests

Table 4.1: Versioning strategies in popular REST API's

As we may see from my short research above, a commonly used way to version API is adding a version number into the URL. The custom MIME type or using additional parameters in URL's is much rarer. What to choose? The answer to this question is not that easy. A lot of

articles in the web (Wood, 2014), (Paraschiv, 2013), (Zazueta, 2015) recommend using Media Type versioning (going throw Content Negotiation) but Subbu Allamaraju in (Allamaraju, 2010) calls to avoid introducing new media type for each version since it leads to media type proliferation, which may reduce interoperability with other server/clients. Peter Williams in his blog (Williams, 2012) completely disagrees with this opinion. I am supporting the idea with using a detectable pattern in the path, e.g.: <http://www.example.com/api/v0.4/>

4.2 Processing a Client Request: Service Layer

The inner layer between any resource and *datahandler* is the *service* module. A resource delegates a request to its own service to retrieve data from a database. After successful response from *datahandler* service executes a private method *getUnwantedExports()* to build a set of unwanted associations (redundant references) which should be extracted (nulled) from current object. An example of a processing a client request to the *OrgUnit* resource is shown below.

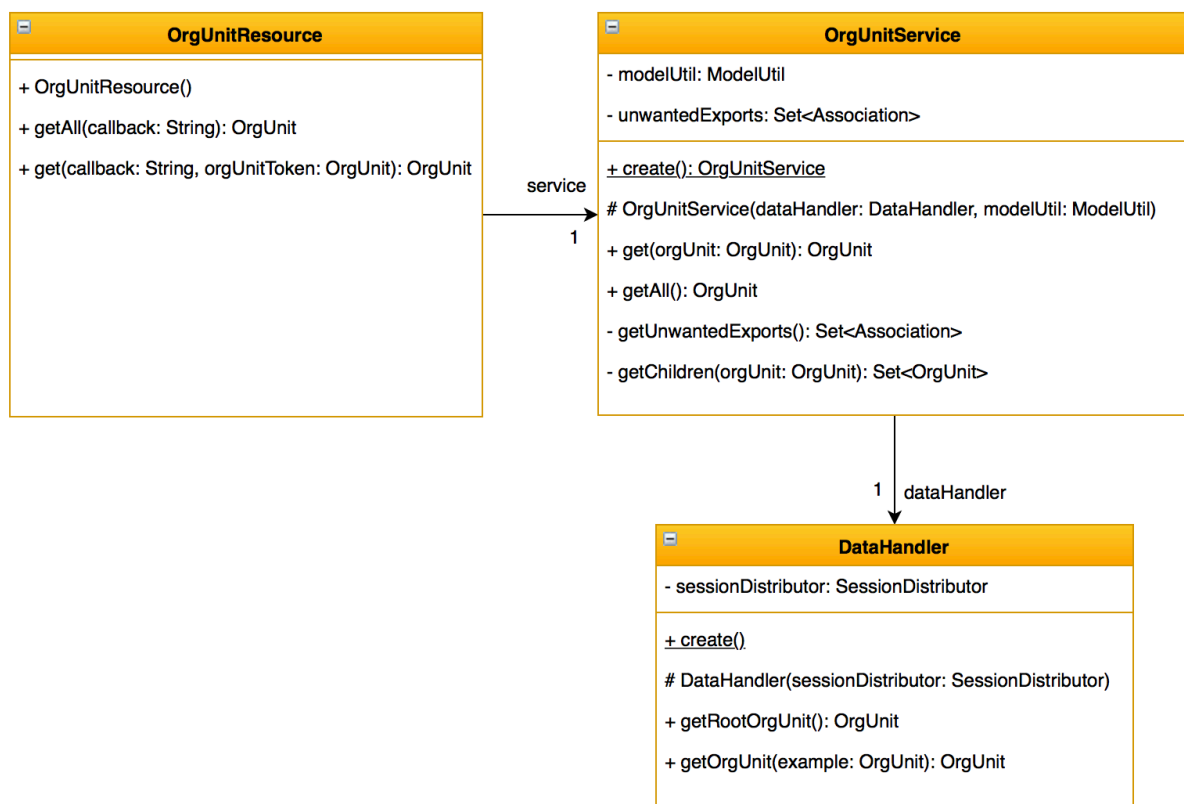


Figure 4.6: Example of a processing a client request

OrgUnit object is a good example to show the problems which will occur without extraction of redundant associations. Each exemplar of *OrgUnit* class contains a reference to its *virtualAccount*. At the same time, a *virtual account* object has a reference to the *orgUnit* object. Already this one bidirectional association causes an endless recursion. Another use case is an elimination of unused relations. For example, an information about assigned members to each organizational unit when it is used for building of a client navigation list is redundant and also should be removed before sending it as a client response.

4.3 Export Utility

The preexisting module **.cmsuite.commons.model.util* provides an implementation for the utility *ModelUtil* to eliminate the redundant associations from a passed object. Within the scope of

my work, I improved the existing development by adding two additional parameters: an association *role* and a *depth*, what is shown in Figure 4.7.

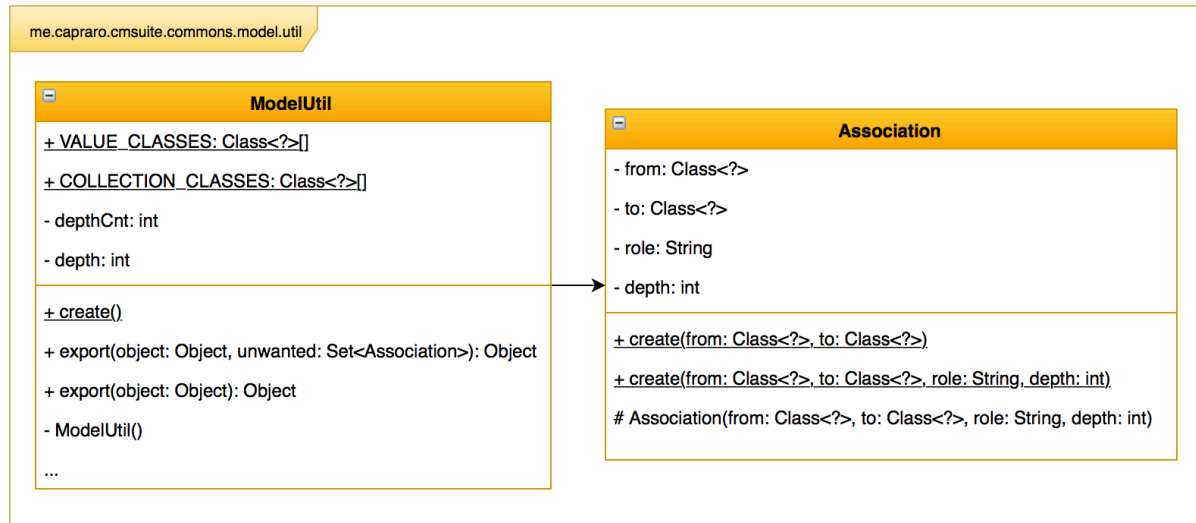


Figure 4.7: ModelUtil

With these parameters, it is possible to manage the entry depth of elimination for specific associations. For example, an absence of “*DailyAccountSummary.class* – *Account.class*” elimination causes an endless recursion by *Account* resource. However, with the extracted association it is not possible to match patches to their contributors and receivers. The solution is a using of a new *Association* constructor and passing a name of association (*role*) and a depth value. It will allow to determine the contributors and receivers and avoid the endless recursion in the next levels of a hierarchical objects’ structure.

4.4 Static Factory Method

My implemented Java classes provide public static factory methods which return an instance of the appropriate class. There are some advantages for a using this way to obtain an instance. First advantage is that the static factory methods do not require to create a new instance each time they are called. If some objects are often requested, a cache- or before created instances can be returned. It can improve a performance significantly. With this method, it is possible to avoid the duplicate objects: we can control what instances are existing at any given time. Another advantage is that a static factory method has a name what makes a class easier to use and do less mistakes.

4.5 Angular Routing

URL Routing within a web application is a common approach for matching the content of the appropriate templates to the specific functionality. As I mentioned in the previous Chapter, I am going to use the alternative to the default module *ngRoute*, the OS routing framework *Angular UI-Router*. It provides a state machine and allows us to define states and manage the transitions between them. With help of this framework we are able to decouple nested states to build a hierarchical tree of them. This way allows to create a layout of any complexity with a little effort. Figure 4.8 shows a state chart with defined multiple states that describes the behavior of a client. This behavior represents a series of events that can occur in this system.

Consider this diagram where the initial state is to be *home*. It represents the homepage of *CMSuite Explorer* from which a user can navigate to the first composite state in this diagram, *analytics*. The template of this state should contain a navigation list and include several nested

states to display the various *Patch-Flow* graphs and diagrams: composite state *graph* is responsible for rendering of bi-directed graphs, *virtual-account* and *account* represent *Patch-Flow* and accounting charts for organizational units and projects, *tree* state renders a hierarchical orgchart diagram. The transitions between states take place on click events.

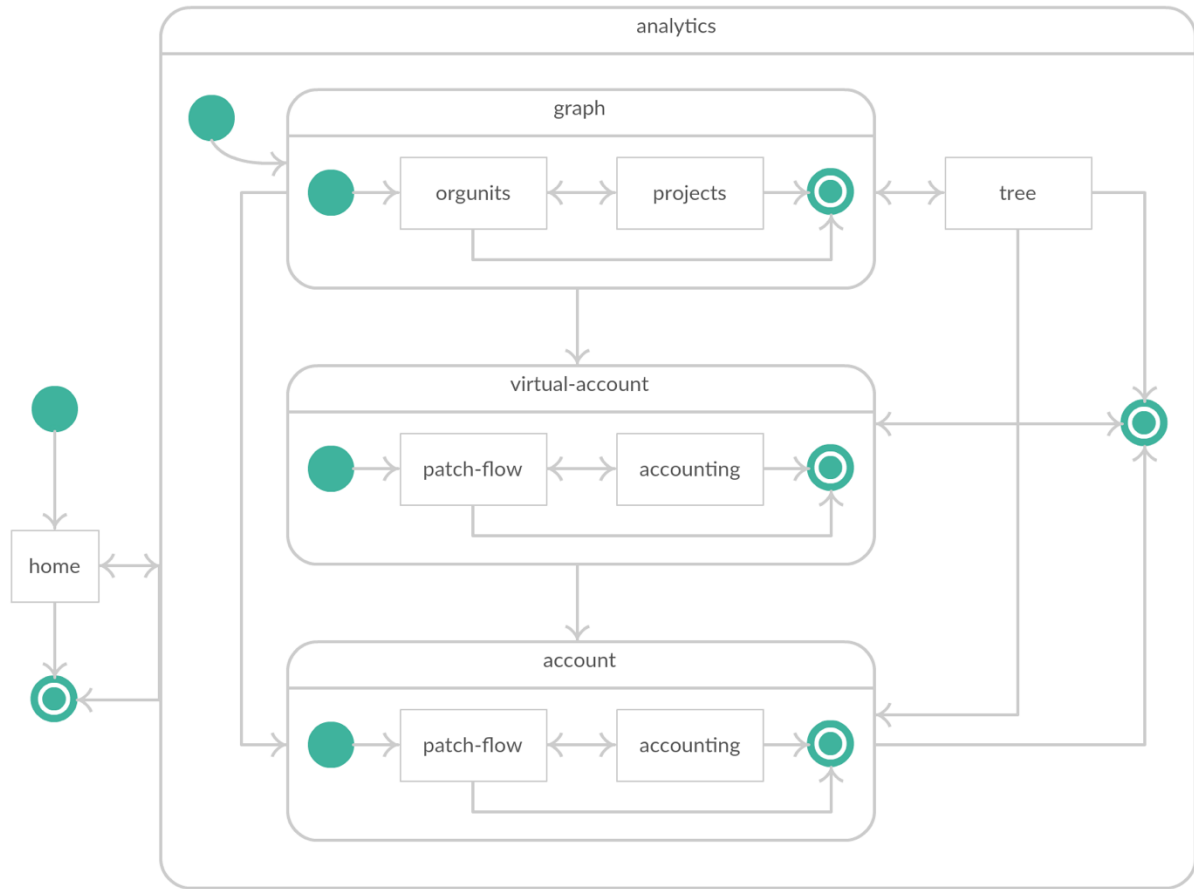


Figure 4.8: State diagram of web client

5 Implementation

5.1 Server with Jersey Framework

5.1.1 Setup with Maven Archetype

According to the official documentation of Jersey (Jersey Documentation, 2016), the most convenient way for working with this framework is to create a new project from Maven archetype. Jersey project is built using a software project management and comprehension tool Apache Maven and runs on top of a Grizzly container (a scalable lightweight HTTP server application in Java).

After a generation of a new project is finished, the created directory *restservice* contains a standard Maven project structure:

- *src/main/java* for all source files
- *src/test/java* for all unit tests

Since Apache Maven supports multiple modules, I created a parent module called *analysisprovider* and included *restservice* as a sub module. Additionally, I created two new packages: a help module *service* and a *datahandler* to retrieve data from database, and also included them to *restservice*. Consider the Jersey package. The source code which is responsible for bootstrapping the Grizzly container, configuration and deployment of this project to the container I placed in *RestServiceApplication.java*.

Next step is a configuration of JSON support. There are different modules that provide support for JSON representations by integrating the individual JSON frameworks into Jersey (Jersey Documentation, 2016). I registered Jackson as JSON provider in *MyObjectMapperProvider* class. To overcome the cross-domain restrictions Jersey provides the support for JSONP which can be added with an appropriate annotation. The following step is an implementation of JAX-RS resources.

5.1.2 Example of Resource Implementation

In Section 4.1.1 I identified 4 resources: *OrgUnit*, *Patch*, *Account* and *VirtualAccount*, which should be available for the client requests.

Consider the first resource from the list, *OrgUnit*. It is responsible for building a hierarchical tree of organizational units at the client and for further navigation within a website. The code snippet shown below is an example of *OrgUnit* resource implementation with Jersey framework. *OrgUnitResource* contains two resource methods annotated with *@Path*, *@GET*, *@Produces* and *@JSONP*. A combined value of class and method *@Path* values builds a relative URI path to the resource method. With *@GET* annotation both methods will process HTTP GET requests. The *@Produces* annotation specifies the multiple media types of resource representation.

The resource delegates the client requests further to the service to retrieve data from the database and provides an export. In the service class I defined the redundant associations and added them into a set of associations (Code snippet 5.2). Then the utility *export()* method is invoked.

```

@Path("/v0.4/orgunits")
public class OrgUnitResource {

    private final OrgUnitService service;

    public OrgUnitResource() {
        service = OrgUnitService.create();
    }

    @GET
    @JSONP(queryParam = JSONP.DEFAULT_QUERY)
    @Produces({"application/javascript", "application/json"})
    @Path("/")
    public OrgUnit getAll(@QueryParam(JSONP.DEFAULT_QUERY) String callback) {
        OrgUnit result = service.getAll();

        if (result == null) {
            throw new WebApplicationException(404);
        }
        return result;
    }

    @GET
    @JSONP(queryParam = JSONP.DEFAULT_QUERY)
    @Produces({"application/javascript", "application/json"})
    @Path("/{orgUnitToken}")
    public OrgUnit get(@QueryParam(JSONP.DEFAULT_QUERY) String callback,
        @PathParam("orgUnitToken") String _orgUnitToken) {

        OrgUnit exampleOrgUnit = OrgUnit.create().setToken(_orgUnitToken);
        OrgUnit result = service.get(exampleOrgUnit);

        if (result == null) {
            throw new WebApplicationException(404);
        }

        return result;
    }
}

```

Code snippet 5.1: OrgUnitResource

```

private Set<Association> getUnwantedExports() {

    Set<Association> unwanted = new HashSet<Association>();

    // No endless recursion from VirtualAccounts
    unwanted.add(Association.create(VirtualAccount.class, OrgUnit.class));
    unwanted.add(Association.create(VirtualBalance.class, VirtualAccount.class));

    // No relations from Persons
    unwanted.add(Association.create(Person.class, OrgUnit.class));
    unwanted.add(Association.create(Person.class, Project.class));

    // No endless recursion from Accounts
    unwanted.add(Association.create(Account.class, Project.class));
    unwanted.add(Association.create(Balance.class, Account.class));

    // Only children, not parents of OrgUnit
    unwanted.add(Association.create(OrgUnit.class, OrgUnit.class));

    // No endless recursion from Projects
    unwanted.add(Association.create(Project.class, OrgUnit.class));

    return unwanted;
}

```

Code snippet 5.2: Helper method for retrieving a list of redundant associations

```

public class OrgUnitService {

    private final DataHandler dataHandler;
    private final ModelUtil modelUtil;
    private final Set<Association> unwantedExports;

    public static OrgUnitService create() {}

    protected OrgUnitService(DataHandler _dataHandler, ModelUtil _modelUtil) {}

    private Set<Association> getUnwantedExports() {}

    public OrgUnit get(OrgUnit _orgUnit) {

        OrgUnit orgUnit = dataHandler.getOrgUnit(_orgUnit);

        if (orgUnit == null) {
            return null;
        }

        orgUnit = orgUnit.setChildren(getChildren(orgUnit));
        return (OrgUnit) modelUtil.export(orgUnit, unwantedExports);

    }

    public OrgUnit getAll() {}

    private Set<OrgUnit> getChildren(OrgUnit _orgUnit) {}
}

```

Code snippet 5.3: OrgUnitService.java

5.2 Server Testing

For powerful testing and verifying the correct implementation of server-side components Jersey provides its internal test tool. Jersey test framework is primarily based on JUnit and helps with designing and running the tests.

5.2.1 Configuration

To add some advanced configuration and features to Jersey test framework, I created my own subclass *RestServiceTest* to override two methods of super class *JerseyTest*. The short code snippet below shows these methods. The overridden method *configure()* enables test traffic logging as well as dumping the HTTP message entity as part of the traffic logging (Jersey Documentation, 2016). Since I use Jackson instead of default MOXy JSON provider, I registered it in the client configuration *configureClient()*. Each resource test inherits this configuration from *RestServiceTest* class.

```

public abstract class RestServiceTest extends JerseyTest {

    @Override
    protected ResourceConfig configure() {
        enable(TestProperties.LOG_TRAFFIC);
        enable(TestProperties.DUMP_ENTITY);

        return RestServiceApplication.createApp();
    }

    @Override
    protected void configureClient(ClientConfig config) {
        config.register(new JacksonFeature()).register(MyObjectMapperProvider.class);
    }

}

```

Code snippet 5.4: RestServiceTest.java

5.2.2 Unit Testing with Mocks

I used a combination of two mock frameworks for writing unit tests in Java: Mockito and PowerMock. Mockito enables to stub method calls and mock objects to verify an object behavior during a test. PowerMock extends Mockito with the powerful features and, for example, enables to stub the static methods. The code snippet below shows an example of using both frameworks during the unit testing of account resource.

```
@RunWith(PowerMockRunner.class)
@PrepareForTest({AccountService.class})
public class AccountResourceTest extends RestServiceTest {

    private AccountService mockedService;

    @Before
    public void setUpMock() throws Exception {
        mockedService = PowerMockito.mock(AccountService.class);
        PowerMockito.whenNew(AccountService.class).withAnyArguments().thenReturn(mockedService);
    }

    /*
     * Tests regarding account list
     */

    @Test
    public void testGetAll_JSONPPresent() {
        List<Account> results = new ArrayList<>();
        PowerMockito.when(mockedService.getAll()).thenReturn(results);

        WebTarget target = target();
        String js = target.path("/v0.4/accounts/list").queryParam("__callback", "JSON_CALLBACK")
            .request("application/javascript").get(String.class);

        Mockito.verify(mockedService).getAll();
        assertTrue(js.startsWith("JSON_CALLBACK("));
    }
}
```

Code snippet 5.5: Example of using mock frameworks for unit test of account resource

5.3 Angular Client

5.3.1 Application Setup

The client is based on an application skeleton, *angular-seed*, which is typically used for a quick bootstrapping of new AngularJS projects. The seed is preconfigured to install the Angular framework and various tools for development and testing. For development I used the following tools:

- *Node.js* with its *npm*: node package manager to get the tools according dependencies
- *Bower*: client-side code package manager to get the angular code
- *Protractor*: end to end (E2E) test runner

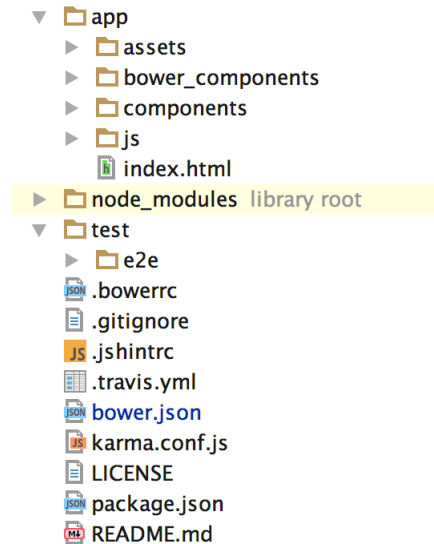
Bower and Protractor are the standard tools in *angular-seed*.

There are two kinds of dependencies in this project: tools to help manage and test the application, and angular framework code. After cloning the *angular-seed* repository using Git I installed dependencies: to get the tools I depend on with help of *npm* and the angular code via *Bower*. All necessary packages for *Bower* configuration are similar to Node's *package.json* and defined by a manifest file *bower.json*. Registering packages allows to install them quickly. After launch the preconfigured *npm* with other scripts I got 2 new folders in my seed project:

- *node_modules* containing the *npm* packages
- *app/bower_components* containing the angular framework files

5.3.2 Directory Layout

Angular seed provides with a modularized structure what actually belongs to the best practices for building scalable and maintainable AngularJS applications. I took the angular-seed project as a basis for my working directory and applied some changes to improve it for my case. The result is shown in the following figure.



Code snippet 5.6: Directory layout for AngularJS project

The directory structure consists of two main folders: *app* and *test*. The folder *app* contains all source files for application, the folder *test* contains End-to-End (E2E) configuration file and scenarios with test cases. The main html template file of the app is *index.html*. Consider the folder *app* in detail. The *components* subfolder will contain all app specific modules. All JavaScript files (*controllers.js*, *directives.js*, *services.js*) and the main application module *app.js* I decided to place into the common subfolder *js*.

5.3.3 MVC Pattern

The power of Angular is to make the static web pages of my application dynamic. I used the MVC design pattern to structure the code in this app: to separate code from views. The data model is instantiated in a simple constructor function we call a *controller*. It takes a *\$scope* parameter, a glue between model, templates and views what allows them work together. I created a new separate file *controllers.js* to keep all the controllers in one module *myAppControllers*. The first controller we are defining is *AnalyticsCtrl*. After his declaration we should register it in the controllers' module and include in our layout template. We also need to add the name of this module to the list of dependencies of main module *myApp* in *app.js*. After providing controller with the data, it will bind the data between model and view. Whenever the model changes, Angular updates the view. The view component is constructed from the template *analytics.html* which is found in the folder *components*. I used this pattern to implement other controllers similarly.

5.3.4 Routing

In this step I provided the routing functionality. I'm getting this framework via Bower by adding the dependency to the *bower.json* and including *angular-ui-router.js* in my *index.html*. I also have to add '*ui.router*' to the main module's list of dependencies. *app.js* holds the root module of application and should be also added with a `<script>` tag to the *index.html*.

The most significant benefit of this routing framework is nested states and views. We turned the *index.html* template into a layout template what means we use this template as a common template for all states in this application.

I defined the app states in the previous Chapter. Next step is an implementation of the first states of this application: *home* and *analytics*. They both should have own view templates. The role of added *ui-view* directive in *index.html* is to include the view template for the current state into the layout template. Similarly, as with the default Angular routing, I used a provider of the route service *\$stateProvider*. With UI-Router it focuses purely on states. I set up the defined two states in the module config as in the following:

```
app.config(function($stateProvider, $urlRouterProvider){
  // set up the states
  $stateProvider
    .state('home', { // register a state config under a given name
      url: '/',
      templateUrl: 'components/home/home.html'
    })
    .state('analytics', {
      abstract: true,
      url: '/analytics',
      templateUrl: 'components/analytics/analytics.html',
      controller: 'AnalyticsCtrl'
    })
  })
```

Code snippet 5.7: Example of implemented client states

\$stateProvider wires view templates, controllers and the current URL location in the browser together. The state *analytics* fetches data and renders the navigation list.

The other important things are *ui-sref* directives. I use them instead of usual *href* attributes of *<a>* tags. They manage state transitions and automatically generates the *href* attributes.

Since I have multiple metrics, it is redundant to place them all on one page *analytics.html*. So, it makes sense to use the advantages of nested states. The templates I already defined can also have their own *ui-view* directives. During design, except *analytics* state, I designed three other combined states: *graph*, *virtual-account* and *account*. Of course, with *ui-router* it is also possible to group the nested states around their parent or abstract parent state. The abstract state is a state which cannot get activated itself and cannot be transitioned to. However, it can have children states and activated implicitly. To map a combined state with its children in *ui-router* a dot notation in form: *parent.child* is used. I grouped two states for *Patch-Flow* graph (*analytics.graph.orgunits* and *analytics.graph.projects*) into the abstract state *analytics.graph*. For detailed *Patch-Flow* information and the accounting ledger are responsible the next four states: *analytics.account.patch-flow*, *analytics.account.accounting*, *analytics.virtual-account.patch-flow*, *analytics.virtual-account.accounting*. These states are also grouped and have own group parents. I also implemented the additional state *analytics.tree* to display the organizational chart. Code snippet below shows all defined states in my app.

```

app.config(function($stateProvider, $urlRouterProvider){
  // set up the states
  $stateProvider
    .state('home', {"url": '/'...})
    .state('analytics', {"abstract": true...})
    .state('analytics.graph', {"abstract": true...})
    .state('analytics.graph.orgunits', {"url": '/orgunits'...})
    .state('analytics.graph.projects', {"url": '/projects'...})
    .state('analytics.tree', {"url": '/tree'...})
    .state('analytics.virtual-account', {"url": '/virtual-accounts/:orgUnitToken'...})
    .state('analytics.account', {"url": '/virtual-accounts/:orgUnitToken/accounts/:projectToken'...})
    .state('analytics.virtual-account.patch-flow', {"url": '/patch-flow'...})
    .state('analytics.virtual-account.accounting', {"url": '/accounting/:virtualAccountId/:interval'...})
    .state('analytics.account.patch-flow', {"url": '/patch-flow'...})
    .state('analytics.account.accounting', {"url": '/accounting/:accountId/:interval'...})
}).run(function ($rootScope, $state) {...});

```

Code snippet 5.8: Client states

5.3.5 REST Client

There are different ways to fetch data from server in Angular. One of them is a low-level service *\$http*. It communicates with the remote HTTP servers via JSONP or other formats. The much easier way a definition of own services that represent a RESTful client. The RESTful functionality is provided by Angular in the *ngResource* model what does not belong to the core of Angular. All REST services I moved to the separate file *services.js*.

Fist service I implemented is *OrgUnitResource* for *analytics* state. The Angular *\$resource* service from module *ngResource* allows us to create a RESTful client with a few lines of code. *OrgUnitResource* service provides an access to the *OrgUnitResource.java* on the server and fetches data about organizational units.

```

var appServices = angular.module('myAppServices', ['ngResource']);

appServices.factory('OrgUnitResource', ['$resource', function($resource) {
  return {
    getOrgUnits: $resource('http://localhost:8080/webapi/v0.4/orgunits', {}, {
      getJSONP: {method: 'JSONP', params: {__callback: 'JSON_CALLBACK'}}
    }),
    getOrgUnit: $resource('http://localhost:8080/webapi/v0.4/orgunits/:orgUnitToken', {}, {
      getJSONP: {method: 'JSONP', params: {__callback: 'JSON_CALLBACK'}}
    })
  };
}]);

```

Code snippet 5.9: Example of REST client

I used Angular's *dependency injection* (DI) to provide the service to the *AnalyticsCtrl* controller. The REST client makes a JSONP request to the server to retrieve all organizational units. The server responds by providing data in a JSON file. Angular framework detects and parses it itself. This service returns a "future" object, which will be filled with data when the response returns. In usual case we would assign the retrieved object to the scope controlled by this controller and bind it to the template. Then, when the data arrives, the view will be automatically updated. However, to modify the fetched data and create my own structure with it assigning to *\$scope.orgUnits*, I passed a callback function to handle the asynchronous response.

```

OrgUnitResource.getOrgUnits.getJSONP(function(response) {...}, function (error) {
});

```

Code snippet 5.10: Callback function

After data is ready, I need to present it in form of navigation list. Since I have a hierarchical set of data (a list of organizational units with its projects) I need a mechanism to render it as a tree. Ben Foster in his article (Foster, 2014) described how to do it with Angular: with so called recursive templates and *ng-include* directive. So I started by displaying the top level of organizational units, the root.

```
<ul class="cmsuite-navtree-group cmsuite-navtree-root">
  <li ui-sref-active="active" ng-repeat="orgUnit in orgUnits" ng-include="'orgUnitsNode.html'"></li>
</ul>
```

Code snippet 5.11: Rendering of a navigation root

To render the next levels of orgunits I used a template and rendered it recursively for each level in the tree. In the following code snippet, I defined an inline template.

```
<!-- A template to render it recursively for each level in the tree -->
<script type="text/ng-template" id="orgUnitsNode.html">
  <a ui-sref="analytics.virtual-account({orgUnitToken: orgUnit.token})"> {{ orgUnit.name }} </a>
  <ul class="cmsuite-navtree-group" ng-if="orgUnit.children.length > 0">
    <li ui-sref-active="active" ng-repeat="orgUnit in orgUnit.children" ng-include="'orgUnitsNode.html'"></li>
  </ul>
</script>
```

Code snippet 5.12: Inline recursive template

For each organizational unit I displayed its name and then a nested list of its sub units by rendering the same template using *ng-include*. What is lacking is a project list which I added for each organizational unit in the next step.

```
<!-- A template to render it recursively for each level in the tree -->
<script type="text/ng-template" id="orgUnitsNode.html">
  <a ui-sref="analytics.virtual-account({orgUnitToken: orgUnit.token})"> {{ orgUnit.name }} </a>
  <ul class="cmsuite-navtree-project-group" ng-show="orgUnit.projects.length > 0">
    <li ng-repeat="project in orgUnit.projects">
      <span class="glyphicon glyphicon-folder-close" aria-hidden="true" style="color:#777"></span>
      <a class="text-muted" ui-sref="analytics.account({orgUnitToken: orgUnit.token, projectToken: project.token})">
        {{ project.name }}
      </a>
    </li>
  </ul>
  <ul class="cmsuite-navtree-group" ng-if="orgUnit.children.length > 0">
    <li ui-sref-active="active" ng-repeat="orgUnit in orgUnit.children" ng-include="'orgUnitsNode.html'"></li>
  </ul>
</script>
```

Code snippet 5.13: Inline recursive template with a project list

Other RESTful services in this application have been written in the same way.

5.3.6 Force-directed Patch-Flow Graph

One of the most significant features providing by CMSuite Explorer is a force graph of *Patch-Flow* between organizational units or projects.

I modified the source code and placed it to my own directive *forceGraph*. According to official documentation of AngularJS (AngularJS Documentation, 2016), directives are markers on a DOM element that tell AngularJS' HTML compiler to attach a specified behavior to the DOM element. When Angular bootstraps the application, the HTML compiler traverses the DOM matching directives against the DOM elements. There are different directive types. The best practice is using directives via tag name and attributes which I applied for *forceGraph*.

5.3.7 Testing

There are two kinds of tests in Angular application: unit tests and End to End (E2E) tests. Due to the time constraints, I implemented only E2E tests.

End to End Testing

The E2E tests are written in Jasmine and run with the Protractor E2E test runner. It uses native events and has special features for Angular applications. Protractor is a Node.js program built on top of WebDriverJS (Selenium WebDriver). It simulates a user interaction, running in a real browser, and verifies that the application responds correctly. This E2E test framework allows to test Angular-specific elements without any setup effort (Protractor, 2016).

Protractor needs two files to run: a configuration file *protractor-conf.js* and a spec file with tests *scenarios.js*. The configuration file contains different information for Protractor: how to set up the Selenium Server, which tests to run, how to set up the browsers, etc. Selenium WebDriver supports several browser implementations or drivers. This application is tested with Google Chrome. Protractor supports two test frameworks out of the box: Jasmine and Mocha. I used the default test framework Jasmine without extra downloading dependencies and set up Chai for Mocha. In my app I placed all tests in one test file *scenarios.js*. With growing complexity of application, it makes sense to organize the test cases in several spec files like *home-spec.js*, *patch-flow-spec.js*, *accounting-spec.js*, etc.

Separating controller from the view in MVC makes it easy to test code. The *describe* functions are used for a grouping related test cases to the test suites. They are nested and define specs at each level. *it* blocks are made of *commands* to tell the framework to do something with the elements of application and *expectations* to assert. If an *it* block fails, Protractor does not stop the execution of other test cases, it just marks the *it* and continuous on to the next block. In the test cases I also used a global *beforeEach* function, which run the code before each *it* block.

Conclusions and Future Work

In this thesis, I developed a web application for visualization of the *Patch-Flow* concept with help of graph and various diagrams. I chose a client-server model and reviewed the web server- and client architecture. I made the evaluation of various OSS components for implementing this architecture and diagrams:

- web server framework for implementing RESTful web services in Java
- client-side framework for building SPA in JavaScript
- client-side routing framework for SPA
- table module to easily display the accounting data with a set of built-in features
- different chart frameworks

I designed the REST-API for developing web services where I identified all resources, chose their representation formats and media types, designed URIs, chose the versioning type on the basis of provided web research. During the development, I solved the problem with endless recursions and redundant associations for retrieved objects from database and introduced the design for improvements of the export utility to fix it. I defined all routing states of a web client and illustrated them on the state chart.

I specified the configuration of Jersey test framework and described how I performed the tests and which mocking frameworks I used. I made an example of implementation of client routing and REST client. I also described how I implemented the *Patch-Flow* graph. By testing, due to the time constraints, I focused on only E2E-tests.

In summary, the implemented tool *CMSuite Explorer* is a first prototype allowing visualizing *Patch-Flow* and various *Patch-Flow*-based metrics. It enables:

- a quantification of the code-level collaboration
- a quantification of an organizational unit's stake in the code-level-collaboration
- managers to evaluate and make informed decisions about the code-level collaboration

between organizational units or projects within an organization. This tool helps users (managers) to analyze the collaborative process and make the software development process within companies more effective.

Future Work

In the future work, a force bi-directed *Patch-Flow* graph can be replaced by another type of the graphical representation, a bi-directional Sankey diagram, in which the width of the links between nodes is shown proportionally to the flow quantity. This can be, e.g. helpful for users to determine the dominant contributions in the flow.

With help of Angular Smart Table framework, a client side pagination for accounting can be added. This framework also provides a powerful pipe/ajax plugin for a paging transition.

References

- Allamaraju, S. (2010). *RESTful Web Services Cookbook*. Sebastopol, CA, USA: O'Reilly Media, Inc.
- AngularJS Documentation. (2016, April 29). *Guide to AngularJS Documentation*. Retrieved from AngularJS: <https://docs.angularjs.org/guide>
- Argefalk, P. J., Fitzgerald, B., & Stol, K.-J. (2015). *Software Sourcing in the Age of Open: Leveraging the Unknown Workforce*. Springer.
- Black Duck Open Hub. (2016, April 15). *Compare Projects (JavaScript-based frameworks)*. Retrieved from Black Duck Open Hub: https://www.openhub.net/p/_compare?project_0=AngularJS&project_1=Backbone.js&project_2=Ember.js
- Bostock, M. (2015, 05 19). *Force Layout*. Retrieved from d3 Wiki: <https://github.com/mbostock/d3/wiki/Force-Layout>
- Bostock, M. (2016, 02 8). *Mobile Patent Suits*. Retrieved from Mike Bostock's Blog: <http://bl.ocks.org/mbostock/1153292>
- Capraro, M., & Riehle, D. (2016, April). Inner Source Definition, Benefits, and Challenges. *ACM Computing Surveys*.
- Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. *Doctoral dissertation*. Irvine, CA, USA: University of California.
- Foster, B. (2014, May 4). *Ben Foster's Blog*. Retrieved from AngularJS recursive templates: <http://benfoster.io/blog/angularjs-recursive-templates>
- Gaic, D. (2015, June 9). *Top 8 Java RESTful Micro Frameworks*. Retrieved March 7, 2016, from Gajotres.net: <http://www.gajotres.net/best-available-java-restful-micro-frameworks/>
- GitHub. (2016, March 13). *GitHub search results sorted by number of stars*. Retrieved March 13, 2016, from <https://github.com/search?o=desc&q=stars%3A%3E1&s=stars&type=Repositories>
- Google Developers. (2015, December 1). *Organization Chart*. Retrieved March 14, 2016, from Google Developers: <https://developers.google.com/chart/interactive/docs/gallery/orgchart>
- Gupta, V. P. (2014, September 1). *Survey of restful web services frameworks*. Retrieved March 7, 2016, from Slideshare: <http://de.slideshare.net/vpgmck/survey-of-restful-web-services-frameworks>
- Jersey Documentation. (2016, March). *Jersey 2.22.2 User Guide*. Retrieved from Jersey: RESTful Web Services in Java: <https://jersey.java.net/documentation/latest/index.html>
- Oracle Documentation. (2010). *RESTful Web Services Developer's Guide*. Retrieved March 09, 2016, from Oracle: <https://docs.oracle.com/cd/E19776-01/820-4867/6nga7f5ml/index.html>
- Paraschiv, E. (2013, July 30). *Versioning a REST API*. Retrieved November 18, 2015, from Baeldung: <http://www.baeldung.com/rest-versioning>
- Protractor. (2016, April 30). *Protractor: end to end testing for AngularJS*. Retrieved from Protractor: <http://angular.github.io/protractor/#/>
- Riehle, D., Capraro, M., Kips, D., & Horn, L. (2015). *Inner Source in Platform-Based Product Engineering*. Erlangen: Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU).
- Shaked, U. (2014). *Airpair*. Retrieved from AngularJS vs. Backbone.js vs. Ember.js: <https://www.airpair.com/js/javascript-framework-comparison>
- Spark. (2015, April). *Spark survey*. Retrieved March 7, 2016, from <http://sparkjava.com/news.html#sparksurvey>
- TechEmpower. (2016, February 25). *Web Framework Benchmarks*. Retrieved March 7, 2016,

- from TechEmpower: <https://www.techempower.com/benchmarks/#section=data-r12&hw=peak&test=fortune>
- Williams, P. (2012, June 25). *Media Types and Profiles*. Retrieved November 18, 2015, from <http://barelyenough.org/blog/2012/06/media-types-and-profiles/>
- Wood, T. (2014, September 15). *How are REST APIs versioned?* Retrieved November 18, 2015, from Lexical Scope: <http://www.lexicalscope.com/blog/2012/03/12/how-are-rest-apis-versioned/>
- Zazueta, R. (2015, March 11). *The ultimate solution to versioning REST APIs: Content Negotiation*. Retrieved November 18, 2015, from The RESTed NARWHL: <http://www.narwhl.com/2015/03/the-ultimate-solution-to-versioning-rest-apis-content-negotiation/>