RWTH AACHEN UNIVERSITY
Chair of Information Systems & Databases
Prof. Dr. Matthias Jarke

A Declarative Web Framework for the Server-side Extension of the Multi Model Database ArangoDB

Computer Science Master Thesis
by Lucas Dohmen
Matr.-Nr. 290333
October 19, 2014

Supervisors: PD Dr. Ralf Klamma, AOR
Chair of Information Systems & Databases
RWTH Aachen University

Prof. Dr.-Ing. Stefan Edlich
Chair of NoSQL, Big Data and Software Engineering
Beuth Hochschule für Technik Berlin

Advisors: Dr. Frank Celler
ArangoDB GmbH
Michael Hackstein
ArangoDB GmbH

Declaration

I herewith declare with my signature, that I have written this master thesis

A Declarative Web Framework for the Server-side Extension of the Multi Model Database ArangoDB

on my own, that all reference or assistance received during the writing of the thesis is stated completely, and that any citation is referenced to its source truly.

_____ Aachen, October 19, 2014

Lucas Dohmen

Contents

1 Introduction

1.1 Motivation

The World Wide Web evolved from the presentation of static content over dynamically created content to machine readable content. With the last step it has become possible for third parties to extend the functionality of a Web page or integrate it with other services via so called Web APIs. Developers could leverage existing APIs to outsource functionality but at the same time the need to create APIs for their own applications arose. The following examples outline some of the functionality that Web APIs offer:

- The API of social networks like Facebook[1] or Twitter[2] allows to build alternative clients for the network, but also get information about the underlying graph.
- Freebase[3] allows structured access to data from Wikipedia.
- Different providers like Facebook, Twitter or GitHub[4] are identity providers for the authorization standard OAuth 2.0 with the option to give the external application access to their information.

Furthermore the capabilities of browsers have become much more sophisticated and the speed of the built-in JavaScript engines increased. This lead to the emergence of so called "single page web applications" – websites that have usage patterns much closer to traditional desktop applications. Handling a lot of logic on the client side, they reduce the need for server-side logic and hence it becomes feasible to create a much simpler server-side application which is only responsible for data storage and security critical sections of code like handling authentication.

Backend systems also evolved with NoSQL systems challenging the existing SQL databases. When Carlo Strozzi in 1998 [Stro98] introduced his database which did not have an SQL interface, he used the term *NoSQL* to describe it. Later in 2009 Johan Oskarsson organized a conference and named it *NOSQL* – coining it as a term for systems that also did not use SQL as their query language. Today, a number of new database management systems are classified as NoSQL systems that do not save the data as relations. Most NoSQL systems do not provide Atomicity, Consistency, Isolation, Durability (ACID) compliance like traditional relational database management systems (RDBMSs) do, but instead focus on a higher read/write throughput or more

[1] `https://developers.facebook.com`
[2] `https://dev.twitter.com`
[3] `http://www.freebase.com/`
[4] `https://github.com`

flexible data models. Some of these NoSQL databases allow extending the database with programming languages like JavaScript.

In this thesis we will look at the NoSQL database ArangoDB. It includes a framework called Foxx that allows the user to enhance the Web API of the database with the help of JavaScript. This is useful for two scenarios:

- When a single page web application only needs a very thin backend as described above.
- When server-side application logic should be moved into the database. This is useful when multiple applications access the same database to prevent duplication or when an operation is data intensive therefore requiring a lot of communication with the database.

1.1.1 Thesis Goals

In this thesis we will present a new design approach and implement a declarative framework for generating web APIs that follow best practices and support the notion of hyperlinks. With this framework we want to answer the following questions:

- Does a declarative approach help to speed up the development of the API? Can it help in keeping the API closer to the problem domain that it acts in?
- How should a Web framework be structured that only allows to define Web APIs? Should it be done differently than in a traditional Web framework?
- We also want to look at best practices when designing Web APIs and existing standards in this area. How can the framework support the developer in implementing those best practices?

The declarative framework will generate a Foxx application. Foxx needs to be extended to be a good target by addressing two missing features: OAuth 2.0 support and background tasks. Both will be implemented.

2 State of the Art

In this section we will first introduce the notion and importance of Web APIs and a classification thereof. We will examine how domain driven design can help with the design of a RESTful API and how it can leverage NoSQL. Finally we will look at the NoSQL database ArangoDB and the status of its built-in Web API development framework Foxx at the beginning of the thesis.

2.1 Web APIs

Richardson and Ruby describe a Web API or Web Service in [RiRu07] as an application intended to be accessed by machines rather than human beings, that uses the technologies from the World Wide Web like the HTTP application protocol, the URI naming standard and the XML markup language.

When designing such an application, the architect can choose from two opposing philosophies of API design: One being Remote Procedure Call (RPC) for example in the form of XML-based SOAP[1] or JSON-based JSON RPC[2] and the other one being Representational State Transfer (REST) as described by Fielding [Fiel00]. Web services following the REST architecture style are often referred to as RESTful Web APIs. It is important to notice that this style is rather a set of recommendations than a standard. We will use the term *RESTful API* in its original meaning going back to Fielding. Pautsasso and Wilde [PaWi09] described the difference between SOAP-based and REST architectures as follows:

- SOAP: Prescriptive approach "assuming closed worlds and contractual relationships". The focus is on integration.
- REST: Descriptive approach that "caters to an open world with ad-hoc interactions". The focus is on cooperation.

In 2010 Maleshkova et al. analyzed 222 Web APIs from the ProgrammableWeb directory [MaPD10]. They found that 85.6% of the APIs offered XML [xml08] exclusively or as one of the output formats while 42.2% offered the much younger standard JSON [json14]. Those two formats were by far the most used output formats in their analysis. In 2012 Renzel et al. [RSKl12] introduced 17 characteristics of a RESTful web service based upon Fielding's recommendations, features of the HTTP protocol and the characteristics described by Richardson and Ruby [RiRu07]. They selected

[1] `http://www.w3.org/TR/soap`
[2] `http://www.jsonrpc.org`

the 25 most popular RESTful Web services from `programmableweb.com` at that time and checked for the adherence to these characteristics. In their analysis we can see the following shortcomings in most of the APIs:

- 75% of the services had no formal description of the web service. Relying only on informal descriptions increases the effort on the user side both in the initial implementation as well as when adapting to changes to the service.
- 80% of them had no links in the representation.
- 95% did use neither XHTML forms nor URI-templates in their representation.
- 35% overloaded a single representation which basically makes them RPC-style APIs.
- 35% encode status information in the response rather than using HTTP status codes.

These results show that the interpretation of Fielding's work vary. As a lot of RESTful architectures have more in common with RPC-style APIs, this distinction can also be made between RESTful APIs and APIs that are a mixture of RPC and REST. RESTful APIs are much closer to this notion: The links between different entities have equal importance to the entities themselves. In the same way we look at the World Wide Web as a graph, we can then look at our API as a graph. But opposed to the Web we are connecting machine readable endpoints that have to be modeled in the application domain of our API. This allows us to decouple the consumer of the API from the ongoing development of the API.

2.2 Use Cases for Web APIs

We described the different kinds of Web APIs. In the following sections we will introduce two important use cases for Web APIs and devise a list of requirements for a framework that helps creating Web APIs.

2.2.1 Single Page Web Applications

As browser technologies became more and more powerful, developers started to develop Web applications that behave more like classic desktop applications. Mikowski et al. [MiPo13] describe this class of Web applications as "Single Page Web Applications". Where early pages of this kind where mostly implemented with Java applets or Adobe Flash/Flex, nowadays JavaScript is fast enough to take over their role. Mikowski et al. name the following advantages of JavaScript over the other technologies:

- It doesn't require a plug-in and works natively in the browser.
- A JavaScript application requires less resources than a Java applet or a Flash application.

Framework	Adoption Readiness	Value Proposition	Votes	Stars	Forks	Contributors
Angular.js	80%	85%	1,498	22,576	7,310	754
Backbone.js	79%	74%	1,299	17,602	3,867	223
Knockout	74%	73%	805	4,838	802	40
Ember.js	68%	74%	676	9,842	2,124	340

Table 2.1: Comparison of MV* frameworks (Last check: April, 11. 2014)

- Both of the other technologies need JavaScript to communicate with the browser. This overhead and additional language is cut out in the case of a pure JavaScript application.

As JavaScript applications became more sophisticated developers started to implement client-side JavaScript frameworks to help organize the applications. This leads to a high number of frameworks which have certain characteristics in common and offer the following base classes:

- A *model* class
- A *view* class or template system
- A *controller*, *view controller* or *view model* class

There's no consistent term for this group of frameworks, we will therefore refer to this group as **MV*** frameworks as each of them has models and views, but the third component differs from framework to framework. InfoQ conducted a survey of the usage of these frameworks[3] with over 2600 votes. The participants where asked to rate the value proposition and the adoption readiness of the frameworks. We took the top four frameworks from this comparison and ranked them according to the results from the survey. As each of them is open source and developed on the source code hoster GitHub, we added the metrics provided there in table 2.1. In the thesis we will look at the three most prominent MV* frameworks according to their GitHub contributors and popularity: Backbone.js[4], Ember.js[5] and AngularJS[6].

Even though a lot of the functionality is implemented on the client-side, certain aspects of the application still need to be addressed on the server-side. On the one hand there is security critical code which can't be executed on the client as it is not a trusted environment. On the other hand data needs to be synchronized between client and server for saving it into the database or exchanging data with other clients. Each of the before mentioned frameworks has functionality to communicate with a Web server via a built-in Web API.

[3] `http://www.infoq.com/research/top-javascript-mvc-frameworks`
[4] `http://backbonejs.org`
[5] `http://emberjs.com`
[6] `http://angularjs.org`

We will describe each of these in detail in section 3.1. What they all have in common is that they expect to communicate with a RESTful Web API. In traditional Web applications the frontend and backend where developed as one project with a strong coupling between the two components – for example with a backend that generates the HTML output for the frontend. With single page Web applications consuming an API HTML is created on the client-side only, the API simply acts as a data source.

2.2.2 Using External APIs

The access to APIs from other companies and giving access to your own data becomes more and more important as described by Tibco [Tibc14]. The consultancy *thoughtbot* regularly releases a document describing the way they work and the tools they use called the "thoughtbot Playbook" [Thou13]. We will use this document as an example on how an application with a lot of external services is built. *thoughtbot* build their Web applications mostly with Ruby on Rails using a long list of external service providers:

- They host their Web applications on the **platform as a service** [MeGr09, p. 2] *Heroku*.
- Their source code is externally hosted on the git-based **source code hoster** *GitHub*.
- All code is run on the **continuous integration service** provided by *Travis CI*.
- The main **database** used is PostgreSQL which they don't host themselves but rather use *Heroku Postgres*. When using Redis, the service *Redis-to-go* is used.
- **Uptime and performance monitoring** of the application is done with the service provided by *NewRelic*. Error Tracking is done via *Airbrake*.
- They use the **transactional email** sending service provided by *SendGrid*.
- **Log analysis** is done by *Splunk*.
- **Payment processing** is done by *Stripe*.
- **Analytical data** is sent to *Segment.io* which then send it to other services to analyze them.

Most of these services were traditionally operated in-house. In the case of the described setup, the Rails applications reports to external services via their APIs. For example if an exception occurs, it will be reported to the API provided by Airbrake, all generated logs will be forwarded to Splunk and if a mail should be delivered, a request to SendGrid is send. Each of those reports is done via HTTP to the API of the according service provider. Other than the outsourcing of the described service, they also outsource bookkeeping and the hosting of emails. This signals that Thoughbot outsources work that traditionally would be done in-house from hosting to bookkeeping.

This allows the team to concentrate on their core competencies. Everything outside of it is done by other parties and this communication is mostly done via HTTP APIs.

Providing an API that should be used by different companies means that consumers of the API have to react to breaking changes to it. One approach to this is to formulate a Service-level agreement (SLA) to guarantee that the API stays the same for a given time frame. This is a model used in the setting of closed worlds and contractual relationships. In this case SOAP is often a better fit. In the case of open APIs this is however not possible as there is no contract between the provider and user of the API. Links between entities, URL templates and forms can be used in this case to reduce the number of breaking changes. It also provides the possibility to remove the links to certain resources when they are temporarily not available as pointed out by Richardson and Amundsen [RiAm13].

Using an API from external services on the other hand requires the developer of an application to react to outages and other failures of this external API. This can be accomplished by putting work with external APIs into a queue with retry functionality. Using a queue will decouple incoming requests to the application from slow or error-prone external services and allow to retry a job later if it could not be finished due to an external API being temporarily unreachable.

We will discuss the creation of a developer API in section 3.2.

2.3 An Approach to Designing RESTful APIs

Richardson and Amundsen [RiAm13] introduced a seven step design procedure to create an API based upon the REST principles:

1. List all entities that should be part of the API including their attributes.
2. Use the descriptors to draw a state diagram of the API. An example from Richardson and Amundsen [RiAm13] can be found in figure 2.1.
3. Try to adjust the attribute names and link relations to existing profiles like IANA[7]-registered link relations.
4. Choose or define a media type.
5. Write a profile that documents the application semantics.
6. Write the code to implement the API.
7. Announce the “billboard URL”, the URL for a start resource that leads to all other resources.

This approach leads to a RESTful API that is well documented. The graphical representation as a state diagram resembles closely the view of the API as a graph. The approach doesn't provide guidelines to design the entities themselves. An approach to design them is introduced in the next section.

[7] `https://www.iana.org`

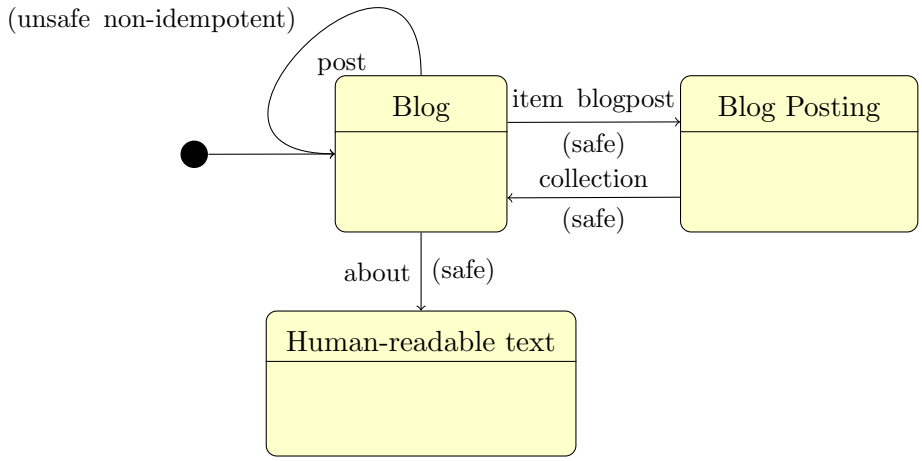

Figure 2.1: State diagram example from Richardson and Amundsen [RiAm13, p175]

2.3.1 Domain Driven Design

When we design our API as described above, we also have to design the resources provided by it. Evans [Evan03] describes an approach to software architecture that focuses on the domain rather than technical details first. A consistent model can be created by finding an ubiquitous language that everyone in the team understands. The language consists of vocabulary from the domain experts and modeling becomes a shared effort between them and the developers. The model consists of domain models that are connected to each other via associations. Evans suggests multiple categories of domain models that we will introduce in the following paragraphs.

Entities are domain models that have an identity. Even if certain attributes of an entity change, they still remain the same entity. An example for this category is a person: The name of a person can change through marriage or the address could change when moving to a new home. In both cases the person remains the same. This is different in the category of **value objects**: They are only defined by their values and have no identity beyond that. An address is an example for that: If someone changes the street number of an address it would be a different address. This distinction leads to an interesting result: Entities are mutable and value objects are immutable.

If a certain set of multiple entities and value objects belongs together to form a domain object we talk about an **aggregate**. In most cases an aggregate consists of an entity combined with multiple value objects. An example would be a person that is connected with the address she lives at. Because value objects are immutable we can use denormalization to store them multiple times. This is possible, as a change to one value object does not have an effect on another value objects with the same value. This is a problem in SQL databases as they don't have a notion of embedding entities

in other entities. We will address this problem in section 2.4.

Furthermore Evans describes **services** as stateless models that are identified by what they do. For example the functionality of sending an email does not require state and could be executed by a mail sending service. A **factory** is a domain object that creates other domain objects. This is especially useful when the creation of the model is really complex. The last category is the **repository** which is a collection of other domain models. It can be thought about as a list with the functionality of adding, removing and querying containing models. This is typically backed by a database.

Each of these domain model types could be a resource in a RESTful API. The associations between the models are the hyperlinks between the resources. When we want to store our domain models in a relational database, Evans suggested "compromising some formal relational standards, such as normalization, if it helps simplifying the object mapping". We will introduce a different approach to both the storage of aggregates as well as modeling the associations in the section 2.4.

2.4 The multi-model database ArangoDB

NoSQL describes a wide area of databases. Sadalage and Fowler [SaFo12] categorize them as either aggregate oriented databases or graph databases.

Aggregate oriented databases lift the restriction described in section 2.3.1 and allow to save an aggregate as one document in the database. Depending on the kind of database, these documents are either opaque or can be queried – which can be accelerated by setting indexes on them. Furthermore, the structure of the document does not follow a strict schema as it would in a relational database, but allows arbitrary attributes and nesting. With nesting we have a different way to store aggregates than we have in a relational database. A value object can be stored as an attribute of an entity for the reasons described in section 2.3.1. We refer to this practice as *embedding* a value object.

Graph databases allow to model data as vertices and edges. The vertices (and in most cases the edges) can be annotated with arbitrary attributes. The user of the database can write queries with respect to the structure of the graph – we refer to this practice as *traversing* the graph. Instead of using joins with foreign keys or join tables, we can model our associations as edges between our vertices.

Each of these categories can help to model in a way that is much closer to the domain than it would be in a relational database, because it does not require both mental and programmatic mapping between the domain models and the database. In the next section we will introduce a multi-model database which combines an aggregate oriented database with a graph database that allows us to model our data in the same way we modeled our domain.

ArangoDB is an open-source NoSQL database system developed by ArangoDB GmbH[8] that combines an aggregate oriented and graph database. We refer to this combination as a **multi-model database**. The schema is determined automatically and is not provided by the user while allowing the user to model the data as either documents, key-value pairs or graphs. ArangoDB is multi-threaded and memory-based: Only the raw data is frequently synchronized with the file system while supporting data is only stored in memory. Due to Multiversion Concurrency Control (MVCC) documents are not deleted – instead, a new version of the document is stored which allows parallel read and write actions. ArangoDB allows to set indexes on the attributes (including nested attributes) with a range of indexes including a hash index (useful to search for equality), skip lists (for range queries) and a geo index. There are multiple ways to access the data in ArangoDB:

1. The REST interface provides simple measures to create, access and manipulate the data using the document-identifier or query-by-example.
2. The Arango Query Language (AQL) is inspired by SQL, but has a slightly different syntax due to the dynamic nature of the document store. Documents can be constructed by the `return` statement and nested documents can be queried. The statements are transmitted via HTTP from the application to ArangoDB and the format of the response is JSON.
3. Furthermore *JavaScript* is embedded in ArangoDB using the V8[9] engine. The documents, AQL and all parts of the API are directly accessible from JavaScript. With the built-in Foxx framework described in section 2.4.1 this custom functionality can be exposed as a RESTful interface.

The graph functionality provided in ArangoDB was first implemented by Dohmen et al. [Dohm12] and later enhanced to feature traversals written in JavaScript and ArangoDB's query language AQL. Hackstein et al. [Hack13] added graph visualization in the administration interface of ArangoDB. In section 2.4 we have described how a document store allows us to model entities, values and aggregates and how a graph database allows us to model the associations in a very natural way. ArangoDB allows us to do both.

2.4.1 ArangoDB Foxx

Foxx is a framework that allows the user to enhance and adapt the HTTP API of the NoSQL database ArangoDB with the help of JavaScript. It was developed in 2013 by Dohmen et al. and is designed to run either behind a server-side application or to be directly accessed by a Web client. Foxx is not a generic framework – it can only be used to create a Web API using JSON as the exchange format. Both the database and the framework are released as open source software under the Apache 2 license. The framework provides three classes that can be used to build web applications:

[8] `http://arangodb.com`
[9] `https://code.google.com/p/v8`

- The **Foxx.Controller** reacts to incoming HTTP requests on parameterized routes with user defined JavaScript functions. At its core it is a *Page Controller* [SaFo12, p. 333], but it automatically translates the JSON body and parameters of the requests to *Foxx.Models*. It therefore also handles the view part which is the translation to JSON.
- A **Foxx.Model** is a *Domain Model* [SaFo12, p. 161] that is ignorant of both HTTP and the persistence to a database. It holds the data provided by either the *Foxx.Controller* or *Foxx.Repository* and can add convenience functions on top of that.
- The **Foxx.Repository** is a *Repository* [SaFo12, p. 322][Evan03, p. 147] – a wrapper around the collection objects of ArangoDB (following the before mentioned repository pattern of domain driven design). It can save to, retrieve from and query the database. All results are automatically wrapped in a *Foxx.Model*, all save operations expect a *Foxx.Model*.

Additionally the routes to the controllers are stored in a configuration file called `manifest.json`. It is used to generate a **router** object. The **collection** object provided by ArangoDB expects and returns JavaScript objects. As the Repository wraps this collection object, an Object Document Mapper (ODM) is not necessary. Both are provided by ArangoDB itself and are used by but not a part of the Foxx framework. The interaction between the user of the API, the Foxx components and ArangoDB is described in figure 2.2.

A Foxx application is written entirely in JavaScript where every Controller, Model and Repository is saved in a separate file to disk. Each of the above described classes is designed to be tested in isolation. Foxx supports a subset of the modules available in the Node.js package repository Node Package Manager (NPM), so reusable components can be extracted from a Foxx application and be shared on NPM.

A configurable number of workers will answer each queued HTTP request therefore making it possible to saturate multiple cores of modern CPUs. Foxx does not implement its own HTTP stack but instead uses the one provided by ArangoDB. The database uses a queue where each of the incoming requests is added to. Every worker operates in a blocking manner where it does not jump to a different part of the execution on IO operations.

2.5 Existing Solutions

In 2012 Zuzak and Schreier [ZuSc12] defined a set of requirements for a REST framework and compared a number of existing frameworks to check if they fulfill them. In their comparison they looked at four frameworks – we will only take the server-side frameworks into consideration:

- *Webmachine*[10] by Basho is written in Erlang and published under the Apache

[10] `http://webmachine.basho.com`

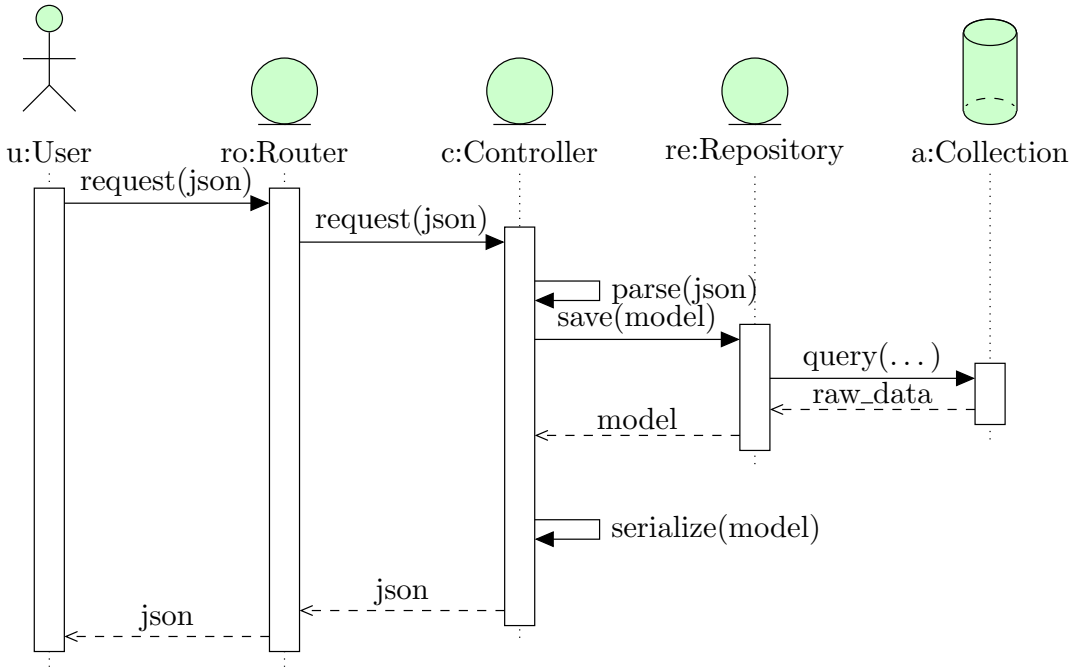

Figure 2.2: UML sequence diagram for current version of Foxx

2 license. It allows to define a set of resources that contain functions over the state of the resource.

- The Java framework *Jersey*[11], an open source implementation of the JAX-RS standard specified in JSR 339 [JSR339] published under the GPL license. With the help of annotations objects can be exposed as resources in a RESTful API. Links between the resources can be added with the help of the framework.
- *Restfulie*[12] is a framework for Ruby, Java and .NET by Caelum Objects released under the Apache 2 license. The authors used the Ruby version for their comparison. The goal of the project is to go past Create Read Update Delete (CRUD) APIs and "provide Hypermedia aware resources".

Zuzak and Schreier state that what is missing for the creation of more Web services that follow the REST architecture are tools that "give developers guidance on following REST principles". Additionally they require the following features from a future solution while none of the evaluated frameworks offer them:

- Research on formal models such as statecharts for modeling resource behavior.
- Allow the user of the framework to define Hypermedia link types.

[11] `https://jersey.java.net`
[12] `http://restfulie.caelum.com.br`

- Support for the extension of supported protocols.

In a step towards modeling APIs as statecharts, Liskin et al. [LSSc12] introduced an UML-based notation for modeling REST services in 2012. It's based upon state machines (or statecharts) as required by Zuzak and Schreier. The description of the service is done in a textual notation. However, the notation doesn't support descriptions of resource structure beyond well-known media types. It's main goal is to plan and visualize the basic design, not to build the service. The notation was not tested with anyone outside of the team. Maximilien et al. [MWDT07] introduced a domain specific language implemented in Ruby to generate APIs in the form of Ruby on Rails applications. The DSL however was designed to create mashups only. Both papers do not offer evaluation with people outside of the team to investigate ease and speed of development.

2.6 Requirements

In addition to the before mentioned gaps in the existing solutions we did a requirement analysis with the users of Foxx 1.0. From this analysis we added the two requirements with the highest demands to the list as they were deemed necessary to use the framework in projects: Support for OAuth 2.0 and the ability to define background tasks that run outside of the request-response cycle. This analysis resulted in the following requirements:

- **Functional Requirements:**
  - Provide a declarative way of defining APIs based upon statecharts or similar models.
  - Provide support for hyperlinks between resources as the transitions of the statecharts.
  - Provide support with the design of the domain of the problem.
  - Provide support with adhering to best practices of API design.
  - Provide support for documenting the API and all introduced media types.
  - Provide support for authentication with external identity providers via OAuth 2.0.
  - Provide support for background tasks that run outside of the request-response cycle.
  - The resulting API is compliant to an existing Web API standard such as JSON+Collection [Amun13] or Siren[13].
  - The resulting API should be resistant to changes in the requirements and reduce the number of breaking changes.

[13] `https://github.com/kevinswiber/siren`

- **Non-Functional Requirements:**
  - The resulting framework should be well tested.
  - Provide manual and documentation for the framework.
  - The code should follow a code style guideline.
  - The framework should be easy to use.

2.7 Summary

We introduced the notion of Web APIs, their use cases and outlined how a combination of domain driven design as described by Evans [Evan03] as well as the approach by Richardson and Amundsen [RiAm13] can be used to design a RESTful API. We then introduced the NoSQL database ArangoDB and the included Foxx framework. Finally we've shown examples for existing solutions to create Web APIs, their shortcomings, derived requirements for a Web API framework from those shortcomings and requirements from existing users of the Foxx framework.

3 Use Cases

We discussed two use cases for web APIs: As a backend for a client-side web framework and as an API for external developers. We will describe those cases in detail here.

3.1 Client-side Web Frameworks

We will take a closer look at three client-side web frameworks: Backbone.js, AngularJS and Ember.js. All of them have built-in capabilities to communicate with a Web API. We will briefly introduce each of the frameworks and then talk about the way they communicate with a Web API and their expectations about the API.

Backbone.js was first released in October of 2010 by Jeremy Ashkenas and is now maintained by DocumentCloud. Compared to the other two solutions, it is much smaller and has less functionality. Backbone.js describes itself as follows:

> "Backbone.js gives structure to web applications by providing models with key-value binding and custom events, collections with a rich API of enumerable functions, views with declarative event handling, and connects it all to your existing API over a RESTful JSON interface." (backbonejs.org)

The communication with the Web API is done via a Backbone Collection which allows the following operations:

- Retrieving, adding, updating and removing entries from the collection.
- Sorting and filtering of the entries.
- Fetching the entries from the Web API and persisting them from the client.
- All collection methods from Underscore.js[1].

Backbone offers the `sync` function[2] whenever it needs to communicate with the Web API. This already implies that Backbone expects the API to follow a collection semantic which is then replicated on the client side. It takes three arguments: The CRUD method (create, read, update, patch or delete), either a model to be saved or a collection to be read and options for jQuery[3]. In listing 3.1 the calls to the public API of the collection and model are mapped to the corresponding calls to `Backbone.sync`. The default implementation of this function uses the `url` attribute and the CRUD method to make the requests (given the URL is `/library`) shown in table 3.1.

[1] `http://underscorejs.org`
[2] `http://backbonejs.org/#Sync`
[3] `https://jquery.com`

Action	HTTP Verb	URL
Create	POST	`/library`
Read	GET	`/library[/id]`
Update	PUT	`/library/id`
Patch	PATCH	`/library/id`
Delete	DELETE	`/library/id`

Table 3.1: Mapping between actions and HTTP verbs and URLs in Backbone's sync method

```
1  var Library = Backbone.Collection.extend({
2    model: Book,
3    url: '/library'
4  });
5
6  var nypl = new Library();
7
8  var othello = nypl.create({
9    title: "Othello",
10   author: "William Shakespeare"
11 });
12
13 // => Backbone.sync('post', othello, { url: '/library' });
14
15 othello.save({ created: 1603 });
16
17 // => Backbone.sync('put', othello, { url: '/library/123' });
18
19 othello.destroy();
20
21 // => Backbone.sync('delete', othello, { url: '/library/123' });
```

Listing 3.1: Mapping between methods and the sync calls

When receiving data from the Web API, it will use the `parse` function of the collection which receives the response from the Web API and returns an array of objects. The default implementation of this function is the identity function. When it sends a model to the Web API it will use the `toJSON` method of the model. In the case of an array of models, it will iterate over the models and call the method on each of them. The default implementation of this function returns all attributes of the model as an object.

The `sync`, `parse` and `toJSON` functions can be overwritten, but have to take the same arguments and return the same kind of output. The semantics therefore will still be those of a collection.

Ember.js was created and is maintained by Tilde Inc. and released in 2011. According to its documentation it has a "sophisticated system for creating, managing and rendering a hierarchy of views that connect to the browser's DOM. Views are responsi-

Action	HTTP Verb	URL
Find	GET	`/library/id`
Find All	GET	`/library`
Update	PUT	`/library/id`
Create	POST	`/library`
Delete	DELETE	`/library/id`

Table 3.2: Mapping between actions and HTTP verbs and URLs in Ember's REST adapter

ble for responding to user events like clicks, drags, and scrolls, as well as updating the contents of the DOM when the data underlying the view changes." It also has mechanisms for routing to move through the different states of the application. According to its documentation it features both controllers and models where "in general, your models will have properties that are saved to the Web API, while controllers will have properties that your application does not need to save to the server". Ember ships with **Ember Data**[4] as its persistence layer. Ember Data supports custom adapters to communicate with the Web API. Those adapters need to fulfill the following interface:

- `find`: Get one record by ID.
- `createRecord`: Create a new record by serializing it and then sending it to the Web API.
- `updateRecord`: Updates an existing record by serializing it and then sending it to the Web API.
- `deleteRecord`: Delete a record.
- `findAll`: Get all records.
- `findQuery`: Get multiple records by executing a query.

Those methods are for example implemented by the included `RESTAdapter` class. It matches the actions to HTTP Verbs and URLs as shown in table 3.2 for a collection `library` and the ID `id`. It has a lot of similarities with the way that the Backbone `sync` method expects the Web API to work except that it does not feature a patch. It again features the semantics of a collection with the conventions for URLs as they are for example implemented by Ruby on Rails.

AngularJS was initiated by Google in 2009. It allows the extension of the HTML vocabulary to create custom components using JavaScript in order to declare dynamic views. It features data bindings to update those views whenever a model changes and adding behavior to those elements via controllers. To communicate with the Web API

[4] `https://github.com/emberjs/data`

it offers a low level HTTP service called `$http` which offers similar functionality to the AJAX functionality of jQuery. The `$resource`[5] factory from the `ngResource` module is built upon that and allows to interact with "a RESTful server-side data source" according to the documentation.

The resulting object features five actions that map to different HTTP verbs shown in table 3.3 which can be extended by custom actions. The actions can either expect to receive an array or a single resource. In comparison to Backbone.js and Ember.js it does not map to a specific URL depending on the action. This is done via the combination of the URL template used to create the resource and the arguments provided to the method. If the provided URL is for example `/user/:userId`, the arguments object provided to the resource object will use the argument `userId` to insert it into the URL. If there is no `userId` argument, it will leave this part blank. That means the requests can go to both `/user` as well as `/user/someId`. From the three frameworks, Angular binds itself the least to the collection semantics.

Action	HTTP Verb	Expect Array
get	GET	No
save	POST	No
query	GET	Yes
remove	DELETE	No
delete	DELETE	No

Table 3.3: Mapping between actions and HTTP verbs in Angular's ngResource

All of these solutions are using collection semantic for the Web API. They treat the API as their database. To create a backend for a single page web application using one of the above frameworks therefore requires to offer it using collection semantics.

3.2 Developer APIs

The access to APIs from other companies and giving access to your own data becomes more and more important as described by Tibco [Tibc14]. Web applications provide access to their data in a machine readable manner so that other applications can be build upon them. This could be a social network that allows others to build clients for a mobile phone, a code sharing site to get statistics for the users and the programming languages they use or simply providing authentication so that users don't have to create another login. The API of the source code repository website GitHub for example allows the following functionality via their public web API[6]:

- Access the activity of users like their notifications and what they are doing on the platform.

[5] `https://docs.angularjs.org/api/ngResource/service/\protect\T1\textdollarresource`
[6] `https://developer.github.com/v3`

- Access the bug tracker of the project with the possibility to open new issues, see existing ones, add comments, close them etc.
- Organize the members of organizations.
- Access the pull requests of a project with the same possibilities as they are present on issues.
- List all repositories of a user, add new ones or delete existing ones. Get an overview of the branches on the repository.
- The entire search functionality of the website including searching for users and repositories.
- Get information about the users of the platform.

If we look at the most popular APIs on ProgrammableWeb[7] we see a wide variety of APIs. The categories of the top ten APIs are the following:

- Mapping (Google Maps)
- Social (Twitter, Facebook)
- Video (YouTube)
- Photos (Flickr)
- eCommerce (Amazon Product Advertising)
- Telephony (Twilio)
- Music (Last.fm)
- Search (eBay)
- Messaging (Twilio SMS)

From those, only YouTube uses collection semantic in their API as the MVC frameworks in the previous section would expect them. As already described by Renzel et al. [RSKl12], non of them feature linking between different resources: The resources are independent from each other. As pointed out by Richardson and Amundsen [RiAm13] those links would increase the resistance to change though as links between entities, URL templates and forms can reduce the number of breaking changes. The links can also be adapted when a resource changes its URL or should not be reached when it is temporarily not available.

Even though some developer APIs may have the collection semantic, we still need more generic standards to allow the providers of APIs to choose depending on their needs.

[7] `http://www.programmableweb.com/apis/directory`

4 Modeling the Domain

We are using JSON as both our medium and our storage format. JSON supports the following atomic data types:

- String: Unicode string known from most programming languages of arbitrary length.
- Number: A floating point number. JSON does not differentiate between integers and floating point numbers.
- Boolean: `true` and `false`
- Null: `null`

It also allows both objects and arrays. An object is a dictionary that maps from strings to atomic data types, arrays or objects. An array can contain any number of atomic data types, arrays or objects. Both of them are not typed so that they can also contain different data types. An array could therefore contain both numbers and strings. Listing 4.1 is an example for a JSON document.

```
1 {
2   "name": "Lucas Dohmen",
3   "birthday": "29.06.1988"
4 }
```

Listing 4.1: Example for a JSON document

As JSON does not support hyperlinks, REST standards need to address this and describe how links are represented. We will describe multiple solutions for that in chapter 6.

For the communication between client and server, we will be using HTTP. Most notable for our modeling are the following aspects:

- The body of both the request and response: As described above both will be represented using JSON.
- The headers of both the request and response.
- The request method of the request.
- The response status code of the response, as specified by the IANA[1].

[1] `http://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml`

We have to take the following request methods into consideration as they are specified in HTTP:

- `GET`: Receive the representation of a resource. It is a safe and therefore does not have any side effects (apart from incidental side effects like logging). The server will respond with a 200 status code if the resource was found and can be sent back to the client and a 404 status code if the resource could not be found.
- `DELETE`: Remove the resource entirely. It is unsafe and idempotent. The server will usually respond with either a 204 to signal success or 200 when there is additional information about the deletion.
- `POST`: Create a new resource. It is neither safe nor idempotent. The server will respond with a 201 if the resource has been created. `POST` is also used for overloading, meaning that it is used to simulate verbs that are not commonly implemented.
- `PUT`: Modify a resource. It is unsafe and idempotent. Typical return values are 200 and 204. Some APIs allow the creation of resources by sending `PUT` to a non existing resource.
- `HEAD`, `OPTIONS`, `CONNECT` and `TRACE` will not be discussed. The first two are used for the exploration of the API and can be added automatically – they do not need to be discussed separately. The other two are only relevant for HTTP proxies.

In the following sections we will discuss how to model the different kinds of Domain objects using JSON and HTTP.

4.1 Entities, Value Objects and Aggregates

As Evans [Evan03] suggested we will embed value objects in entities when they form an aggregate. This is possible in JSON as we can nest objects in other JSON objects. Listing 4.2 demonstrates how we embed a value object in an entity. In the example the entity is a person and the value object is the home address of that person.

```
1 {
2   "name": "Alice",
3   "age": 32,
4   "home_address": {
5     "street_name": "Bobstreet",
6     "house_number": 23,
7     "city": "Cologne"
8   }
9 }
```

Listing 4.2: Example for embedding value objects

In a similar way we can do an embedding of multiple value objects of the same type by using an array of objects. In the following we will therefore only talk about aggregates as we will not store value objects separately and entities are a special case of aggregates where the number of value objects is zero.

If we want to model the relationship between two entities we have two possible ways to do that:

- Embedding: In the same way that a value object can be embedded, an entity can be embedded in a response. In contrast to a value object it needs to be saved separately though.
- Referencing: There is no standard way of linking to another entity in JSON as mentioned above.

This requires us to either have a link to the other resources or embed them in the response. How this is done depends on the media type we are using and will be described in chapter 6. In the database, we will use the graph functionality of the database to connect entities using edges.

In the response from the API we will model our aggregates as JSON objects. In our database we will store them as a document. A state that is an entity has to react to all or the subset of the following HTTP verbs (with the status codes described above):

- `GET`: Get the representation of the entity. This process might include embedding referenced documents or hiding certain parts of the representation.
- `DELETE`: Remove the entity.
- `PUT`: Modify the entity. It is also possible to allow to create a resource with a `PUT`.

The creation of an entity is usually done via a Repository or Factory as described in the next section.

4.2 Repositories and Factories

The repository as described by Evans will be shown as an array of aggregates in our API. From the outside the difference between repositories and factories is not really visible as the user of the API cannot see which documents existed before (stored in a repository) or are new (created by a factory).

As factories can be implemented by a service, we will treat it as a special case of a service. The repository however needs to interact with the database in order to implement its functionality.

From the perspective of the database, a repository is a collection. The collections in ArangoDB provide the functionality to implement adding, removing, modifying and searching for entries. As ArangoDB does not require a schema for the entries of a collection, they can just be saved without further modification into the database.

As the described approach focuses on links, the repository needs to link to the contained entities. It needs to be able to add new entries and search through the entries. Therefore it needs to react to the following HTTP methods with the status codes as mentioned above:

- `GET`: Get the representation of the entries in the collection. As this might be quite a lot of entries, the resulting representation needs to be paginated. The representation of the entities might be a short version of the representation shown when getting the entity itself.
- `POST`: Add an entry to the repository.

4.3 Services

A service can do a variety of things which may or may not include database interaction. We will refer to this as the *action* performed by the service to use the terminology by Harel [Hare87]. We first need to differentiate between two kinds of services in an HTTP API:

- A *synchronous service* which means that the action is performed during the request-response-cycle. The result of the action can therefore be used to generate a response so that it can respond with a 200 status code.
- An *asynchronous service* which means that the action is performed outside of the request-response-cycle. That means that the result of the action can not be used in the response and it can not be determined if the action will be successful. Therefore the return status can only communicate if the action was successfully transmitted. This can be done with either a 204 or, if additional information can be provided, with a 200.

The HTTP method a service responds to is determined by the nature of the action that is performed. Each of the methods is possible, therefore the person implementing the action has to decide for an appropriate verb.

We use JavaScript to formulate the actions as it can be natively executed by our target platform. By providing a JavaScript function with the information about the request and the possibility to set a response, the synchronous service can be implemented as required by the user. In the case of the asynchronous service however, we need to perform the following tasks:

- Parse the request.
- Take all information about the request and convert it into a job description that can be interpreted by the action.
- Inform the user of the API that those steps were successful by a response with a 200 or 204 status code.

- At some later point take the job description and provide the action with all information from it.

The recurring task needs to have information on how many threads should work on fulfilling the tasks. As there might be high and low priority tasks, we introduce the concepts of queues where for every queue can have a custom number of threads.

Because we have a database available, we can store the job description in a collection. This way a recurring task can check if there are job descriptions that have not yet been processed and then start processing them. In the job description we can also store additional information:

- `status`: Has the job been worked on yet or has it failed?
- `queue`: In which worker queue should this job be processed?
- `type`: Which JavaScript function should work on this task?
- `failures`: If the job failed, this can be used to store details on the failure like stack traces.
- `data`: The parameters of the request.
- `backOff`: Between each retry the workers will wait for the time specified here.
- `maxFailures`: When a job failed, it will be retried for a configurable number of times.
- Handlers for success and failure. As described above, they can not influence the response of the request, but those functions can be used for logging etc.
- Created and Modified timestamps.

Listing 4.3 contains an example for a document containing the information described above that can be stored as a document in a collection.

```
{
  "status": "pending",
  "queue": "my_queue",
  "type": "my_type",
  "failures": [],
  "data": {
  },
  "created": "1409844354464",
  "modified": "1409844354464",
  "maxFailures": 5,
  "backOff": 1000,
  "onSuccess": "function() {}",
  "onFailure": "function() {}"
}
```

Listing 4.3: Example for a job description

As services are stateless, we do not need to provide any mechanisms to store information between calls of the same service. Services are only identified by what they execute, therefore we only need to provide the action that should be executed and the information about when the action should be executed.

4.4 Summary

With the resource types described above, we can model our domain according to the suggestions by Eric Evans:

- We can model aggregates (entities with zero or more value objects attached) that can be linked to other aggregates.
- We can model repositories containing this aggregates responsible for their persistence.
- We can model both synchronous and asynchronous services, which are stateless.

5 Statecharts

After discussing the types of resources we can model, we now need to discuss how we want to model the flows through our application. Fielding describes in his thesis [Fiel00] that the application state is stored by the client to avoid the bottleneck of storing all state on the server. He further states:

> "The model application is therefore an engine that moves from one state to the next by examining and choosing from among the alternative state transitions in the current set of representations."
>
> (Architectural Styles and the Design of Network-based Software Architectures, page 103, Roy Fielding)

From the viewpoint of an API consumer the API can therefore be modeled as a state machine or a statechart where the server is ignorant of the current state, so we have to store it on the client. The client is however not aware of the entire statechart, only the current state and the available transitions from there. In other words: The server knows the entire statechart, but not the current state. The client knows the current state, but not the entire statechart. In this chapter we want to discuss how we can use statecharts to model our API.

Harel introduced statecharts in 1987 [Hare87] as an extension to state-transition diagrams to form a highly structured description language. They were introduced in UML 0.8 in 1995 as state diagrams (and again renamed to state machine diagrams in UML 2.0). It addresses the modeling of *reactive systems* which are systems that have to react to external and internal events. It has to model the allowed sequence of input and the according output. As in a statechart the basic modeling follows the form of "when an event `e` occurs while in state `A` and a condition `c` holds then go to state `B`". A state diagram can therefore be visualized by a directed graph where the vertices are the states and the edges are the possible transitions. But Harel extended this approach to address the following requirements:

Allow states to be clustered in *superstates*: Introduce the notion of clusters in the form of boxes that can be drawn around multiple states. A superstate can again have a start state which uses the same notation as the start state of the entire statechart. Edges are allowed to both begin and end at any level. When an edge begins at a superstate it means that every state within that superstate can transition using that edge. When a transition ends in a superstate, then the start state will be reached after the transition.

Allow *independence* and *concurrency* via orthogonality. This means that the statechart can be in more than one state at a time. This possibility is denoted by splitting a cluster into two parts via a dashed line. Each part needs a start state. When a transition goes into the cluster both statecharts will be active at once. Every transition

will be executed on both parts simultaneously. If a transition is not available in one of the clusters, it will only be executed in the other clusters.

Allow more *general transitions* than an event-labeled arrow: A transition can have a condition under which it is possible. A condition is acting as a guard to determine if the transition can be executed as soon as the event is triggered or not. This gives additional options like two transitions from A at event E where only one is fired if and only if the conditions are distinct. In an API this can be used for authentication for example: A transition is only possible if the user is authenticated or if the user has administrator privileges.

Allow both actions along transitions and when entering, exiting or continuously throughout being in a state. This allows side effects when transitioning from state to state like interactions with a database.

Allow to formulate that there is a set of states that have identical internal structure and are differentiable via some parameter. An example for that might be two different kinds of alarms which have different side effects (the sounds they make), but have identical transitions to other states. They can be differentiated by their name for example. Another example would be documents of the same kind with different identifiers. Harel refers to this as *parameterized states*.

5.1 Statecharts and RESTful Web APIs

In the model of a statechart the client needs to know how to get the start state of the diagram. This process is referred to as "announcing the billboard address" by Amundsen [RiAm13]. From that point on the server will provide the client with information about the current state and all possible transitions from this state. The client then uses the data and its business logic to decide which transition should be executed and transfers this information to the server. The server again answers with information about the new state and the transitions from there etc. The client does not know the actions nor the conditions. The server filters the possible transitions from a state and then transmits those in the representation. The client is therefore not aware of transitions for which the conditions are not fulfilled.

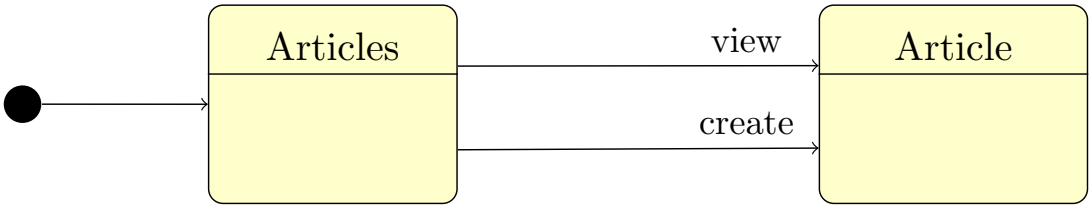

Figure 5.1: Statechart for the Blog API

In the case of a simple blogging API we would have the start state that provides a list of all blog posts. This could be modeled as a statechart as shown in figure 5.1. When the client enters that state it will get a representation that contains both a list of short representations of each of the blog posts and a number of state transitions:

- With the view transition the statechart will get into a new state that shows the full representation of one blog post.
- With the create transition the statechart will get into a new state that shows the newly created blog post in a full representation.

In our API, the states are identified by a specific URI. The client keeps track of the current URI. The transitions are represented by hypermedia links to other URIs. As transitions can be safe or unsafe (and therefore be represented by either GET or POST) and will need to be able to take parameters. We will see different ways of representing those kinds of links in an API in chapter 6.

Using the capability of statecharts to define actions for both transitions and the entry and exit of states, we can implement the required functionality.

Using the possibility of defining conditions for transitions, we can for example integrate authorization. A transition like `destroy` could only be triggered under the condition that the current user is an administrator.

In a lot of APIs there will be collections of analogous items. In our blog example that could be the articles written for the blog. In APIs those will often have similar URIs (for example `/article/id`) and representations. In a statechart we can simulate this using parameterized states – in the example the state is parameterized by the ID of the article. We adjusted the blog API example in figure 5.2 to use a parameterized state for the articles.

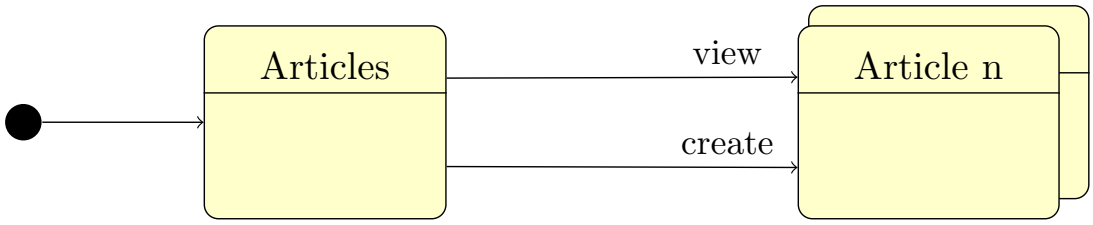

Figure 5.2: Statechart for the Blog API with parameterized state

We also have to pay attention to the possibility of transitions between certain states changing during the lifetime of an API: In the described blog application, a new state could appear when we create a new blog post. In order for this blog post to be reachable by the consumer of the page, it needs to be linked from some other page of the blog. It therefore requires us to create a transition between the other state and the new blog post. In the representation of the other entity, a link has to be added to show the possible transition from there to the blog post.

The API should not allow the creation of arbitrary states and transitions though. Adding or removing of states and transitions needs to follow certain rules that are defined by the creator of the API. We need to be able to both follow a transition as well as defining positions in our statechart where adding or removing them is possible. We also need to be able to allow modifications of states. Therefore the following transition types are required:

- Follow: This is a normal transition that we can follow to get into another state. In HTTP, we can represent this transition by a GET as using it is both safe and idempotent.
- Link: This is a transition that connects two states adding an outgoing link to the representation of the first state. In HTTP, we can represent this transition by a POST as we are changing the first state by appending a link to it.
- Unlink: In this case we remove the link from the representation of the first state. In HTTP, we can represent this transition by a DELETE as we are removing the link from the representation.
- Modify: This is a transition from a state to itself, where following the transition changes the state. In HTTP, we can represent this transition by a PUT or PATCH as we are changing the resource.

It is also important to note that the only states that we can add during the runtime of the application are parameterized states. Otherwise it would not be possible to determine which transitions are possible from this state or which transitions can be added to this state at runtime. Transitions however can be added between both parameterized and non-parameterized states.

5.2 Domain Driven Design in States and Transitions

We will now use the patterns from domain driven design to simplify common patterns of the states:

- **Entity or Aggregate:** As our API uses JSON to communicate with the client, we can natively express aggregates using nested JSON objects. We can therefore treat returning an entity as a special case of the aggregate where our entity is connected to zero value objects.
- **Repository:** If we want to access a collection of aggregates, we could use a JSON array to represent this data structure. The exact representation will depend on the standard that is used in the API.
- **Service:** A service executes code provided in JavaScript and can answer with an arbitrary JSON response.
- **Factory:** Is very similar to an aggregate from the outside as the only difference is that it does not persist the generated object. As its implementation is a special case of the service, we will not implement the factory as a special case.

The repository is a gateway to our database. In the case of ArangoDB, a repository maps to an ArangoDB collection. The API consumer has to be able to perform the following tasks:

1. Return a list of all aggregates stored in the collection. Depending on the amount of data this list should be paginated.
2. Return a subset of all aggregates stored in the collection. It needs to accept some kind of query to fulfill this task.
3. Return a specific aggregate stored in the collection. This can be thought of as a special case of the second task or as a link to an aggregate.
4. Add a new aggregate to the collection.
5. Remove an aggregate from the collection.
6. Update or replace an aggregate.

To fulfill these tasks, the collection needs to be able to map between the JSON objects and documents stored in the collection. As discussed above we can store the representation as is in a document store. Future extensions could allow to hide certain fields, but this will not be implemented in the scope of this thesis. With this information we can use the JavaScript code shown in listing 5.2 in our generator.

With these transition type and parameterized states, we can model the collection semantics of a repository:

- Follow: The user of the API can follow the link from the repository to one of the parameterized states to get the detail view of this item.
- Link: When the user links the repository to a new state, the item described by this state is added to the repository. It now has a link in its representation pointing to the new state.
- Unlink: Unlinking a state from its repository removes the link from the representation and the item from the repository.

When linking two entities we need to store the information that the entities are connected now. As we are using a graph database in our implementation, we can use edges between the entities to save this information. To display all outgoing links of an entity, we iterate over all neighbors of this entity:

- Follow: Get the representation of the linked entity.
- Link: Add an edge between the two entities which adds the links to the respectively other state as a result.
- Unlink: Remove the edge between the two entities which removes the links to the respectively other state as a result.

The transitions to a service will be fixed at design time. We will therefore only allow to create follow links between arbitrary states and a service which will add the link to this service. In the case of a service following the link is not necessarily a `GET` request, but depends on the nature of the service. Therefore the service needs to know which HTTP verb is appropriate to reach it.

5.3 Describing States and Transitions in JavaScript

After we have established which state types we need and how the transitions should work, we need to define how to describe them using a JavaScript Domain Specific Language (DSL). The DSL will then generate an API following the principles described above. We need to be able to define our own custom transition types which can be used by the states. In addition to those custom transition types, a media type that has certain semantics can provide transitions. Furthermore we need to define our aggregates and value objects.

5.3.1 States

To describe our state we will need to provide the following properties:

- What data does the state provide? This results in both the representation returned to the client as well as information about what needs to be stored in the database.
- What transitions are possible?
- Entry and exit actions.

In FoxxGenerator we can describe a state with those properties as shown in listing 5.1. We first need to initialize the generator. Then we add a state with the name `person` and configure it:

- We make it an entity, one of the types of domain objects we described. This specific type needs additional information provided in the form of attributes. The attributes are needed for both the representation and as meta information for transitions (like `link` transitions that create objects following the representation).
- We make it a parameterized state.
- We add transitions from this state to other states. Each of the transitions needs information about the target state (or parameterized state) and which transition it should be. We will describe how to define transition types in the next section.

```
1  // Setup for the generator
2  var FoxxGenerator = require('./foxx_generator').Generator,
3    generator;
4
5  generator = new FoxxGenerator('example', {
6    mediaType: 'application/vnd.api+json',
7    applicationContext: applicationContext,
8  });
9
10 // Add a state named ‘person‘
```

```
generator.addState('person', {
  // This state is of the type 'entity'
  type: 'entity',
  // It is an parameterized state
  parameterized: true,

  // Information necessary for the representation
  attributes: {
    name: { type: 'string', required: true }
  },

  // Add transitions from this state to other states
  transitions: [
    // Provide the name of the other state and the
    // name of the transition
    { to: 'people', via: 'container' },
    { to: 'todo', via: 'assigned' }
  ]
});
```

Listing 5.1: Define a state via JavaScript

A repository on the other hand doesn't need information about the representation (it will be provided with those information by the transitions connecting it to the entities it contains) and will in most cases not be parameterized. Listing 5.2 shows all information necessary to initialize a repository state called `people`.

```
generator = new FoxxGenerator('example', {
  mediaType: 'application/vnd.api+json',
  applicationContext: applicationContext,
});

generator.addState('people', {
  type: 'repository',

  transitions: [
    { to: 'person', via: 'element' }
  ]
});
```

Listing 5.2: Define a repository via JavaScript

5.3.2 Transitions

To define our own transition types we will need to provide the following properties:

- A name for the transition so we can use it in the transitions array of our states.
- A human readable description of the transition
- Information about the arity of the transition: Can there only be one transition from a certain state or can there be multiple ones

- Optionally a condition that checks if the transition is possible. This is represented by an arbitrary JavaScript function which is provided with all information about the HTTP request so that the developer can for example check for a current user. This defaults to a function that always returns true.
- Optionally a precondition which will be used to check if the link should be added to the representation of the start state or not. This defaults to the condition.
- Optionally parameters of the transition. `parameters` is a value object as described in section 5.3.3. Those will be the parameters of the HTTP request.

In JavaScript we can describe a transition with those properties as shown in listing 5.3. We provided three examples: The first example shows a simple `follow` relation which connects a collection of items to the representation of a single item. We then define its inverse transition which will lead back to collection. The knowledge about the relation between the two transitions provides us with important information necessary for the generation of the API and its underlying data model. The third transition uses link semantics to add an idea to a collection. In order to use this transition, the user has to be logged in which is checked by looking at the current user of the request. We also provide information about the parameters needed for the resulting `POST` request.

```
1  generator = new FoxxGenerator('example', {
2    mediaType: 'application/vnd.api+json',
3    applicationContext: applicationContext,
4  });
5
6  // Example for a simple `follow` transition and its inverse transition
7  generator.defineTransition('showDetail', {
8    semantics: 'follow',
9    description: 'Show details for a particular item',
10   to: 'one'
11 }).inverseTransition('showAll', {
12   description: 'Show all items of the same type',
13   to: 'one'
14 });
15
16 // Example for a transition which adds an idea by linking it
17 generator.defineTransition('addIdea', {
18   semantics: 'link',
19   to: 'one',
20   description: 'Add an idea',
21
22   condition: function (req) {
23     return req.session.get('userData') !== undefined;
24   },
25
26   parameters: {
27     title: Joi.string()
28   }
29 });
```

Listing 5.3: Define a transition via JavaScript

By defining the transitions they become available for the states. Their human readable information is used to generate a documentation for the resulting API.

In addition to these custom states a media type can define default transitions. A media type that has collection semantics for example could provide transition types for transitions between a collection and its members and vice versa. This is necessary when these transitions are represented differently from the default ways of linking between states in the media type. The generated API then needs to follow the defined standard.

5.3.3 Aggregates and Value Objects

As described above we can represent aggregates natively in both our representation and in the aggregate-oriented database we are using to save our data. However the developer still needs to be able to describe how the aggregates look like so we can verify their structure before they are saved in the database or used in a service.

A different way to view value objects is to describe them as a custom data type we introduce. Those data types can then be either represented by a native type (for example a date could be represented as a String using the ANSI/ISO format) and an object with different fields. Both types could occur one or several times where in the latter case we would use an array.

When we allow to use custom data types, we need to provide the developer with a way to define them in a format that we can use to verify incoming data. This format should allow different kinds of validations:

- Check for the underlying JSON type (e.g. String, Number, Object, Array)
- Make it required or optional and in the latter case provide defaults.
- Set a minimum and maximum for a number or restrict it to only contain integers and not decimal numbers.
- Use a RegEx to check a String for a certain format or provide a minimum or maximum length.
- Combine value objects into new value objects by representing them as a JSON object.

As this is not the scope of this thesis we decided to use an external library to solve this task. We're using `joi`[1] as it fulfills all the requirements above, works within ArangoDB's JavaScript runtime and is open source. Listing 5.4 shows an example introducing a value object for an address consisting of the street, zip code and house number in Germany.

```
1 var Joi, address;
2
```

[1] `https://github.com/hapijs/joi`

```
3  Joi = require('joi');
4
5  address = Joi.object().keys({
6    street: Joi.string().required(),
7    house_number: Joi.number().integer().required(),
8    house_number_extension: Joi.string().optional(),
9    zip_code: Joi.string().length(5).regex(/^\d+$/).required()
10 });
```

Listing 5.4: Define an address using Joi

If we use the Joi expression above to verify `{ street: 'Ahornstraße', house_number: 55, zip_code: '52056' }`, Joi will provide us with the information that it is valid and with a representation that has all default values filled in. If we provide it with an invalid object like `{ street: 'Ahornstraße', house_number: 55, zip_code: '5205'}` it will return a human readable error explaining that the zip code has the wrong length which the API can then display to the user.

Joi is used in both the description of our entities as well as the description of parameters for requests. In both cases Joi allows to define data types ranging from simple (e.g. the field should be a number) to complex (e.g. an object with strings of a certain format).

6 RESTful Standards

We want to look at existing standards that define RESTful APIs using either JSON or XML as their medium. We will evaluate the standards and propose JavaScript APIs that help generating an API for a subset of them. In our analysis we have to differentiate between different types of RESTful API standards:

- What kind of standard is it?
  - Personal standard: Created by a single person reflecting his or her opinion on how to solve a certain problem. We only take personal standards into consideration, that are well documented.
  - Open standard: The standard has been adopted by a recognized standard body (for example ANSI, ECMA, ISO or the W3C). It has gone through a process of design by committee.
  - Request for Comments (RFC) or Internet draft: This standard is on its way of becoming an open standard. It is first drafted by one or more authors and accepts feedback for six months. After this time it becomes an RFC or the draft is updated – again with a lifetime of six months.
- What format is it using? In the described standards this may either be JSON, XML or both.
- What are the application semantics of this standard? There are mainly two cases:
  - Specific: The standard has specific application semantics.
  - Generic: It does not have a specific application semantic and can be used to build arbitrary APIs.

A simple example for a specific application semantic is a collection. As a lot of APIs are designed to be represented in *collection semantics*, those standards have a lot of different use cases. The collection semantic encompasses adding and removing items to the collection, modifying a single element in the collection and querying for a subset of the elements.

In table 6.1 we collected a number of existing standards and checked wether they met the above mentioned criteria. In the following sections we will look into detailed descriptions of each of those standards.

Standard	Kind	Format	Application semantics	IANA Media Type
Collection+JSON [Amun13]	Personal standard	JSON	Specific: Collections	`vnd.collection+json`
OData[1]	Open standard	XML/JSON	Specific: Collections	Unknown
JSON API[2]	Internet draft	JSON	Specific: Collections	`vnd.api+json`
Hypertext Application Language (HAL) [Kell13]	RFC	XML/JSON	Generic	`vnd.hal+json` and `vnd.hal+xml`
Siren[3]	Personal standard	JSON	Generic	`vnd.siren+json`

Table 6.1: Standards we want to take into consideration

6.1 Link Relations and Media Types

One of the core principles in a RESTful systems are the links between resources. A link consists of two parts: The **URI** where the linked resource can be found and the **relation** between this two resources. The relation is key for the link to be machine readable as it gives meaning to this connection. Therefore the link relations need to be described in the media type used by the API.

The Internet Assigned Numbers Authority (IANA) is a standards body that defines Internet Protocol related symbols and numbers. Part of that function is to define media types and link relations. There are about 70 general purpose link relations[4] which can be used in new media types to simplify the adoption. Some examples for those link relations include:

- `edit`: Refers to a resource that can be used to edit the link's context.
- `item`: The target URI points to a resource that is a member of the collection represented by the context URI.
- `prefetch`: Indicates that the link target should be preemptively cached.
- `up`: Refers to a parent document in a hierarchy of documents.

Relations are also responsible for listing media types[5] such as the image formats. We will later see that most RESTful standards have registered media types with IANA.

6.2 The Collection Pattern

The collection semantic encompasses adding and removing items to the collection, modifying a single element in the collection and querying for a subset of the elements. We will introduce three standards that have this semantics: Collection+JSON, OData and JSON+API.

[1] `http://www.odata.org`
[2] `http://jsonapi.org`
[3] `https://github.com/kevinswiber/siren`
[4] `http://www.iana.org/assignments/link-relations/link-relations.xhtml`
[5] `http://www.iana.org/assignments/media-types/media-types.xhtml`

6.2.1 Collection+JSON

Collection+JSON is a personal standard by Mike Amundsen [Amun13] using JSON that has the specific application semantics of a collection. The following operations can be done with Collection+JSON:

- Reading the entire collection.
- Adding, updating or deleting an item.
- Searching for a subset of the items.

It has similarities to the Atom Syndication Format[6] and the Atom Publishing Protocol[7]. An endpoint of a Collection+JSON API always returns a representation containing different kinds of objects that are children of the root JSON object `collection`:

- The `version` attribute should be set to the current version of Collection+JSON – which is is 1.0.
- The `href` is a link to the collection itself.
- In `links` additional resources can be linked. This element is optional.
- All members of the collection are partially represented and linked to in the `items` array. This element is optional.
- In `queries` there are descriptions of the kinds of queries that can be performed on the collection. This element is optional.
- The `template` object contains all the input elements used to add or edit collection entries. This element is optional.
- The `error` object contains additional information on the latest error condition reported by the server. This element is optional.

An example for a representation featuring each of them – except the error element – can be found in listing 6.1. In the following sections we will go through each of the parts and briefly explain their utility.

```
1 {
2   "collection": {
3     "version": "1.0",
4     "href": "http://example.com/blog.json",
5
6     "items": [
7       {
8         "href": "http://example.com/blog/1.json",
9         "data": [
```

[6] http://tools.ietf.org/html/rfc4287
[7] http://tools.ietf.org/html/rfc5023

```
10            { "name": "title", "value": "My first blog post" }
11          ],
12          "links": []
13        }
14      ],
15
16      "links": [
17        { "href": "/logo.png", "rel": "icon", "render": "image" }
18      ],
19
20      "queries": [
21        {
22          "href": "/search",
23          "rel": "search",
24          "prompt": "Search by title",
25          "data": [ { "name": "title", "value": "" } ]
26        }
27      ],
28
29      "template": {
30        "data": [
31          { "prompt": "Title of the blog post", "name": "title", "value":
                "" },
32          { "prompt": "Content of the blog post", "name": "body", "value":
                "" }
33        ]
34      }
35    }
36  }
```

Listing 6.1: Example for Collection+JSON

The links are an array of objects, each representing a link to a related resource of the collection. Each of them has two required properties (`href` and `rel`) and three optional properties (`name`, `render` and `prompt`).

The items are the entries of the collection. In the case of a detail view of a specific item, this array contains only a single item. In all other cases it can contain zero or more items. Each of the objects again has three attributes: The `href` is the link to the standalone resource, `links` (with the same properties as the top level links) point to other resources related to this specific item and `data` holds information regarding the representation of the item. Data is represented as an array of objects, each with a `name`, `value` and `prompt` field.

The template describes how to create a new entry in the collection: A POST request to the collection has to be sent with values for all fields in data. To find specific items, the template described in queries can be used. In addition to the data field from template, it will require an endpoint to send the request to (`href`, a relation type (`rel`) and a description (`prompt`). The requests are sent via a GET request. Both template and queries are conceptually similar to HTML forms. They provide information about which form fields can be filled.

As most collections will be too large to be displayed entirely in every request, pagination will be needed in a lot of cases. The Collection+JSON standard does not explicitly define how to handle pagination. However, the IANA defines two relations `next` and `previous` that can be used in the `links` parts of a collection to implement pagination.

6.2.2 OData

Similar to Collection+JSON, OData has the application semantic of a collection pattern. It is even closer to the Atom Publishing Protocol as an OData API can also serve an Atom representation.

One important difference between the two standards is the way it handles filtering: While the API in Collection+JSON gives a list of possible queries in the form of hypermedia forms, OData provides a query language for filtering the results. The queries are sent encoded as query parameters. The following parameters that can be specified:

- `$filter`: Restrict the set of returned items by providing criteria. The query can use a set of built-in filter operations including `eq` (equal), `ne` (not equal), `lt` (less than), `and` or `add` (addition). It also features a bunch of built-in functions such as `contains`, `round` or the geometric distance `geo.distance`. An example for that would be `$filter=Name eq 'Milk' and Price lt 2.55` to get all products with the name 'Milk' that also have a price less than 2.55.

- `$orderby`: Specify the order in which the items should be returned, by providing a comma-separated list of expressions such as `Rating desc`.

- `$top` and `$skip`: Can be used to paginate the results, for example `$top=5&$skip=2` would skip the first two results and then only return the next five results.

- `$select`: Only return the explicitly specified properties, dynamic properties, actions and functions. To only get the rating and release date we would for example use the statement `$select=Rating,ReleaseDate`.

The entire set of features of the query language makes it a relational algebra. It closely resembles an URI based version of a language like Structured Query Language (SQL) to query all data that the API provides. In addition to that, OData allows to define custom transitions. They are defined as hypermedia forms and the standard differentiates between two types:

- **Safe** transitions are referred to as *functions* and will be executed with a HTTP GET.

- **Unsafe** transitions are referred to as *actions* and will be executed with a HTTP POST.

Which functions and actions are available, their functionality and what kind of data they expect is described in a Common Schema Definition Language (CSDL) document[8]. CSDL has some similarities to Web Services Description Language (WSDL)[9] – it can even be used to describe the entire service, removing the need for providing the meta data alongside the document as described above. As we concentrate on RESTful services in this chapter, we won't go into this variant of OData. An example for the output of a single entity can be found in listing 6.2

```
{
  "@odata.context": "serviceRoot/$metadata#People/$entity",
  "@odata.id": "serviceRoot/People('lucasdohmen')",
  "@odata.etag": "W/\"488FF11DD1212BC9\"",
  "@odata.editLink": "serviceRoot/People('lucasdohmen')",
  "UserName": "lucasdohmen",
  "FirstName": "Lucas",
  "LastName": "Dohmen",
  "Emails": "lucas.dohmen@rwth-aachen.de",
  "Adresses": [
    {
      "Street": "Aachener Strasse",
      "City": {
        "CountryRegion": "Germany",
        "Name": "Cologne"
      }
    }
  ]
}
```

Listing 6.2: Example for OData

6.2.3 JSON API

JSON API is a personal standard developed by Yehuda Katz (Rails Core Team Alumni[10] and Ember.js Core Team[11]) and Steve Klabnik (Author of Designing Hypermedia APIs [Klab13]). It is interesting as EmberData – the default way of communicating with an API in Ember.js – expects a JSON API compatible API. It also uses collection semantics.

A response is always a document with a key that is the plural form of the resource, the value is an array. This is the case for both collections of documents as well as single documents. While this entry is required it also has three optional top level keys that will be described later.

Each document in the array needs to have an ID that uniquely identifies the document. In addition it may have an attribute `href` that points to the document and

[8] http://docs.oasis-open.org/odata/odata/v4.0/os/part3-csdl/odata-v4.0-os-part3-csdl.html

[9] http://www.w3.org/TR/wsdl20-primer

[10] http://rubyonrails.org/core/alumni

[11] http://emberjs.com/team

`links`. All other keys are attributes of the document. `links` is a JSON object and represents relationships to other documents. The key identifies the kind of resource and the value can have different forms:

- A string or number that represents an ID.
- An array of strings or numbers that represents a list of IDs.
- A link object with one or more of the keys ”id”, ”ids”, ”href” or ”type” (never both ”id” and ”ids”).
- An array of link objects.

The three optional top level keys are the following:

- `meta`: Meta-information about a resource, such as pagination.
- `links`: URI templates to be used for expanding resources' relationships URIs.
- `linked`: A collection of documents, grouped by type, that are related to the primary document(s) and/or each other.

The `meta` section allows to provide certain meta information about the usage of the API. Currently there is only one possible meta information that can be provided by the API: Wether the ID of a document is set by the server or by the client.

The `links` section allows to define URI templates to describe how to get access to related resources. If we take the example of blog posts again, we may want to refer to the author page and to related blog posts. We could do this as demonstrated in listing 6.3. In this example the author of the blog post with the ID 1 can be requested via the URI `http://example.com/blog/1/author`. The second blog post has two related posts. One of them has the ID 21 and can therefore be requested via the URI `http://example.com/blog/21`. It is also possible to request multiple related posts by providing their IDs as a comma separated list, for example both related posts could be requested via `http://example.com/blog/21,22`.

```
1  {
2    "links": {
3      "posts.author": "http://example.com/blog/{posts.id}/author",
4      "posts.related": "http://example.com/blog/{posts.related_posts}"
5    },
6
7    "posts": [{
8      "id": "1",
9      "title": "My first blog post",
10     "links": {
11       "related_posts": [ "10", "20", "30" ]
12     }
13   }, {
14     "id": "2",
```

```
15      "title": "My second blog post",
16      "links": {
17        "related_posts": [ "21", "22" ]
18      }
19    }]
20  }
```

Listing 6.3: Referencing external resources in JSON+API

The `linked` section allows to save roundtrips necessary for the links by embedding the related resources in the response. In this section the resources are grouped by their key (in the example above that would be `related_posts`). The client can request this by appending the query parameter `include` with the key of the resources to be included to the URI.

In addition to that the following other query parameters can be used to influence the output of the API:

- `fields`: Only return certain fields in the response, exclude all others.
- `sort`: Sort the results via one or more attributes.

6.2.4 Summary

If we sum up the standards for collection patterns, we can see that they have a tendency to allow the user of the API to use it as a database accessible via REST. This makes it interesting for client-side applications running in the browser. From the perspective of Evan's pattern language an API following the collection pattern has an object following the repository pattern at its center. It links to a number of entities and each of the formats allows to link to one or more services to use additional functionality.

Our generative framework should therefore allow to generate APIs that follow this pattern. It should be expressed in general terms so it can be extended to support a different collection centric media type in the future.

- The central repository should be transformed into a resource that offers listing a (paginated) view on all or a filtered set of the model in the repository.
- The entity should have a short form (for viewing it as part of a set of models), a long form (for viewing it standalone) and a description of required and optional attributes for a newly created model.
- It should allow to add items to and remove items from the collection in the way it is described in the standard.
- It should allow to create, show, edit and destroy a single model in the way it is described in the standard based on the description of the entity.
- It should allow to define filters in the case of a Collection+JSON media type.

- It should allow to add additional services and allow to link to them. Each service needs to have a description that yields if it is a safe or an unsafe operation.

As JSON+API has the possibility to introduce further links to external entities, we decided to choose it as our candidate for the collection semantic standard.

6.3 Generic Standards

6.3.1 HAL

HAL is an RFC [Kell13] by Kelly that proposes a generic RESTful standard that comes in a JSON and a XML variant. We will introduce the JSON variant here.

The focus of the standard is on the links between resources. The attributes of a resource will be stored as normal attributes in the JSON document. The document has two special attributes: `_links` and `_embedded`.

The `_links` are a section to provide hypermedia links to other resources. The key of each of the entries is the link relation of this link or an array of link relations. The value is an object which needs to have an `href` attribute that contains the URI itself. In addition it may have the following attributes:

- If `templated` is set to `true` then the `href` attribute is treated as an URI template instead of an URI.
- The `type` is a string that indicates the media type of the target resource and the `profile` describes its profile.
- With `deprecation` the link can be marked as deprecated. The value is the URI of an explanation why this link is deprecated.
- The `name` attribute can be used to select one link when multiple links with the same relation are present.
- The `title` is a human readable description of the link.
- `hreflang` indicates the language of the target resource.

The `_embedded` section is similar to the `_links` section only that it embeds the target resource or parts of it instead of linking them. Again it can be both a single object or an array of objects.

One additional feature of the standard is the compact URI referred to as `curie`. It is set as a relation in the `links` section and allows to reduce duplication between URIs. If we for example define a curie for our example blog under the name `blog`, we can afterwards use this in other relations to shorten the URI by prefixing the relation with `blog:`. In the example in listing 6.4 the URI for `next` will expand to .

```
{
    "_links": {
        "curies": [
            { "name": "blog",
              "href": "http://example.com/blog/{rel}",
              "templated": true }
        ],
        "blog:next": { "href": "15" }
    }
}
```

Listing 6.4: Example for a curie in HAL

HAL does not describe how to define the relations. Without this description, the consumer of the API neither knows which HTTP verb to use nor which parameters are allowed. Without the possibility of defining a machine readable description of the relation the author of the API has to write them in human readable form, which then again has to be read by the developer of the API consumer.

6.3.2 Siren

Siren[12] is a personal standard developed by Kevin Swiber that has generic application semantics. In comparison to HAL, Siren gives a lot more structure to both the entities themselves as well as the links between resources. Every entity can have the following attributes:

- The `class` is an array of strings. Siren does not give any details on those strings and leaves details to the implementation.
- The properties of a resource are stored in `properties`.
- Related entities can be embedded via `entities` in the form of an array. They again can have the same attributes as the entity itself. It has an additional required attribute: A link to the parent entity.
- The `title` can contain a descriptive text about the resource. Both are arrays of objects.
- We will go into `links` and `actions` in detail in the next sections.

Siren offers both actions for executing state transitions and links for client navigation. Each link needs to contain a `rel` and a `href` attribute and can contain a descriptive text in form of a `title`. One of those links should be a link to itself. The `actions` are a list of behaviors exposed by an entity. Each action is described by

- The `name`.

[12] `https://github.com/kevinswiber/siren`

- The implementation-dependent `class`.
- The HTTP `method`.
- The `href`.
- The descriptive `title`.
- The `type` of the request defaulting to `application/x-www-form-urlencoded`.
- An array of `fields` for the request. This has a lot of similarity to HTML fields.

Each of the fields has a `name`, a `type` (all HTML5 input types are valid), a `value` and a `title`. An example for the output of a Siren API can be found in figure 6.5.

```
{
  "class": [ "person" ],

  "properties": {
    "name": "Lucas Dohmen"
  },

  "entities": [],

  "actions": [
    {
      "name": "add-relationship",
      "title": "Add a relationship to another person",
      "method": "POST",
      "href": "http://example.com/person/12121/relationships",
      "fields": [
        { "name": "person", "type": "text" },
        { "name": "kind", "type": "text" }
      ]
    }
  ],

  "links": [
    { "rel": [ "self" ], "href": "http://example.com/person/12121" },
    { "rel": [ "brother" ], "href": "http://example.com/person/1111" }
  ]
}
```

Listing 6.5: Example for a Siren API

6.3.3 Summary

For the generic standard we chose Siren, as it features a machine readable format to describe transitions and their arguments. This sets it apart from HAL where the developer still needs to create external documentation resulting in the problems described before. Siren concentrates on the links between resources, which is also the focus of the generator.

7 Implementation

We implemented a declarative JavaScript framework that allows developers to describe the API in terms of the domain using statecharts. Our declarative approach is based upon a combination of Richardson and Amundsen's [RiAm13] design approach described in section 2.3, domain driven design [Evan03] as described in chapter 4 and the statecharts by Harel [Hare87] described in chapter 5. We also addressed the additional requirements regarding OAuth 2.0 and background tasks. An example that covers the use cases outlined in chapter 3 is shown in figure 7.1.

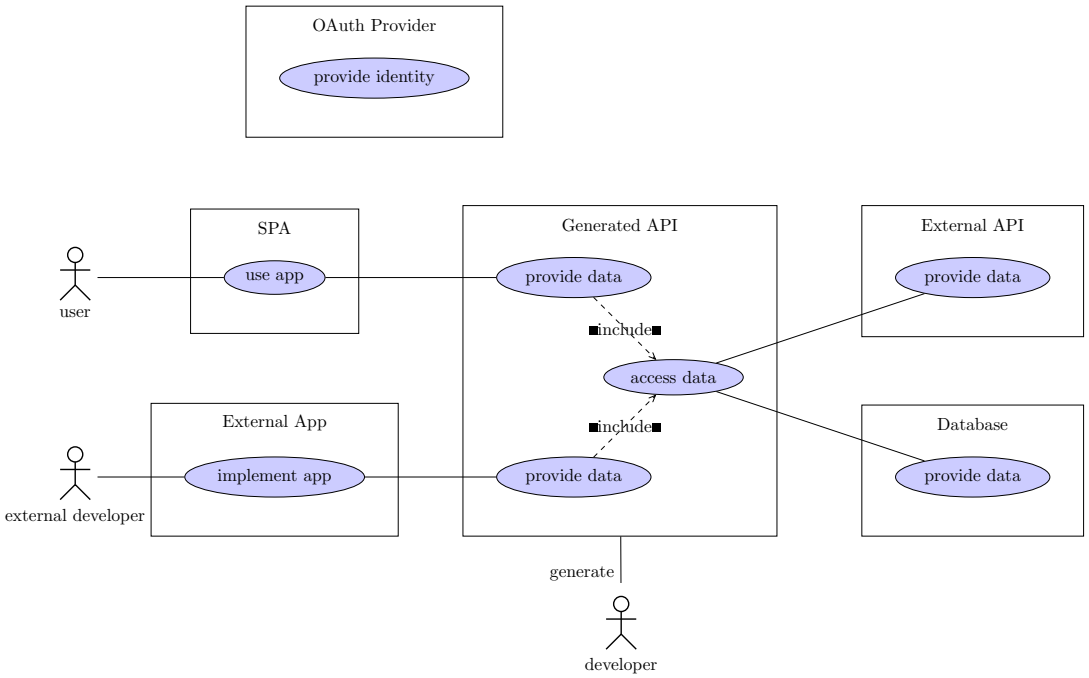

Figure 7.1: UML use case diagram for the generated API

We decided to use the graph capabilities of ArangoDB to model the relations between the individual entities as they are conceptually very close to the underlying statechart. The different kinds of relations are modeled by distinct edge types. This way we can use the neighbor functions to determine hyperlinks for the representation and – if the media type requires us to do so – separate them by relation type. As the edges in ArangoDB are always directed, we can add a link from a resource a to a resource b by adding an edge from a to b and filtering for outgoing edges only when adding the links to the representation of a resource. The declarative framework will generate all

parts necessary for the API to work, in particular:

- The controllers, repositories and models necessary for the Foxx application to run, including the annotations for the automatically generated documentation.
- The collections that are required to save the entities in the database.
- A graph to model the relations between different entities.

7.1 States and Relations

In the framework the domain objects types described by Evans have been implemented as types of states. The states are the resources of the API: Every state knows its representation. The representation is modified by the media type that the user has selected in the generator and can be extended by the transitions to and from the state. In addition to being able to adjust the representation of a state, a transition can also add routes and manipulate the underlying graph of the API. The latter is necessary in order to create relations between two entities.

In order to determine which routes will be added, how the graph will be manipulated and what will be added to the representation of the two states, FoxxGenerator looks at the type of the two states, if the transition is to one or multiple entities and the type of transition (Follow, link, unlink and modify – see section 5.2). This is implemented by using the strategy pattern as defined by Gamma et al [GHJV94]: A strategy has been defined for every valid combination. FoxxGenerator will iterate over each of the strategies and check if the strategy is applicable. If so, the strategy will add the according routes, adjust the representation of one of the states or manipulate the underlying graph. When none of the strategies is applicable to the combination, an error is raised as this combination is illegal. The strategies are implemented specific to the media type.

In addition it will generate the following Foxx classes depending on the type of state for each of the states:

- For every `entity` it will generate a corresponding `Foxx.Model`
- For every `repository` it will generate a corresponding `Foxx.Repository`
- For every `asyncService` it will check if the required `Foxx.Queue` exists and will create it if it is missing.

7.2 FoxxGenerator

Using the DSL as defined in chapter 5 a developer can describe the desired statechart `FoxxGenerator` will use to generate a Foxx application. It will iterate over each of the resources and generate a corresponding `Foxx.Controller` from the resource description using the determined strategy for the selected media type. Each of the outgoing

relations will result in a link to another resource and may alter the representation of the states.

The target media types that have been selected for this thesis are (as described in chapter 6):

- JSON+API as an example for a standard with collection semantics
- Siren as an example for a standard without specific application semantics

The two standards do not support the same feature set. Siren has an additional type of state – the start state. The representation of this state will be used as the root URL of the API. Other entities that should be reachable from there can be connected via transitions. JSON+API doesn't have a general concept of linking an entity that is not a collection or part of a collection and can therefore not offer a start state. Another difference are the relation types that are automatically supported by a media type. Every media type can define a set of media types that do not need to be defined by the user of FoxxGenerator. As JSON+API has collection semantics, it for example supports relations expressing that an entity is an element of another entity. JSON+API only supports links between entities, it however does not allow forms (e.g. transitions with parameters that use a different verb than get) apart from the predefined ones.

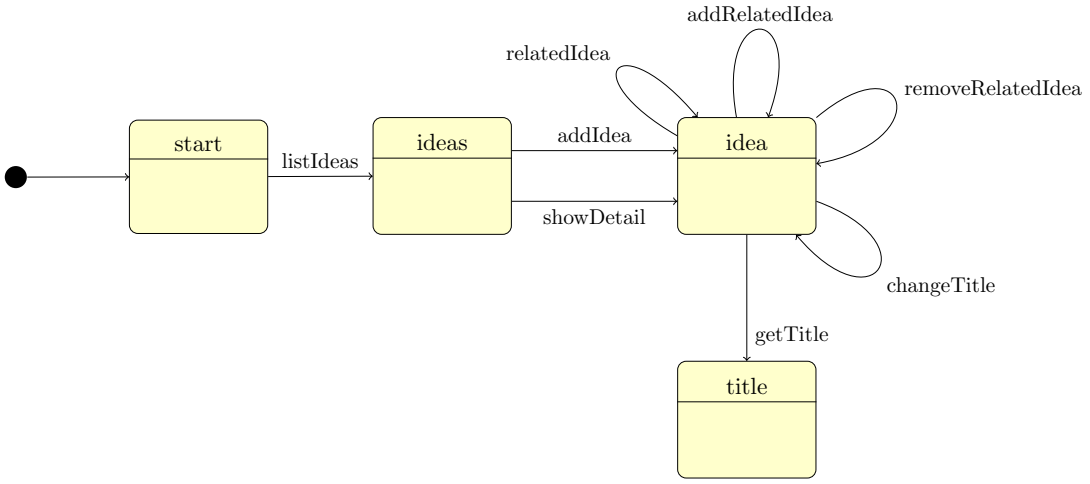

Figure 7.2: Statechart for an app to manage ideas

In figure 7.2 we modeled a simple idea application using a statechart, where users can collect ideas and connect them to each other as being related. It is also possible to change the title of an idea and to only get the title of it. The ideas in the application have a `title` and a `description`. In order to define the statechart in Foxx Generator, we first need to analyze the transitions in order to define them in the DSL of FoxxGenerator:

- `listIdeas` is a follow transition.
- `addIdea` is a link transition as it should allow to link the repository of ideas to a new ones. The transition also needs parameters because the user has to provide a `title` and a `description` to create an idea.
- `showDetail` is a follow transition.
- `relatedIdea` is a follow transition.
- `addRelatedIdea` is a link transition that connects two idea entities. It also needs information about which transition it should create, in this case a `relatedIdea` transition.
- `removeRelatedIdea` is an unlink transition that disconnects two idea entities. It also needs information about which transition it should remove, in this case a `relatedIdea` transition.
- `changeTitle` is a modify transition which connects an idea to itself. To follow this transition, a user has to provide the new title. Therefore the title needs to be a parameter of the transition.
- `getTitle` is a link transition as it allows the user to connect an entity with a service.

With the transitions defined we can now create the states of the statechart. We reanalyze the states:

- `start` is a start state with no additional attributes.
- `ideas` is a repository state with no additional attributes.
- `idea` is a parameterized entity state. It has `title` and `description` attributes which are strings.
- `title` is a service state. We use the verb `GET` as this will only return information without altering the state of the server. We will define an action that extracts the title and returns it.

We can express the transitions and states we found as follows in the DSL shown in listing 7.1. The information are the same that we analyzed above plus a few documentation strings. The generator is configured with the name `ideaapp` and configured to generate an API using the Siren standard.

```
1 var FoxxGenerator = require('foxx_generator').Generator,
2   Joi = require('joi'), generator;
3
4 generator = new FoxxGenerator('ideaapp', {
5   mediaType: 'application/vnd.siren+json',
```

```
6   applicationContext: applicationContext,
7 });
8
9 generator.defineTransition('listIdeas', {
10   semantics: 'follow',
11   description: 'Get the list of all ideas',
12   to: 'one'
13 });
14
15 generator.defineTransition('showDetail', {
16   semantics: 'follow',
17   description: 'Show details for a particular item',
18   to: 'one'
19 });
20
21 generator.defineTransition('relatedIdea', {
22   semantics: 'follow',
23   description: 'Show related ideas of this idea',
24   to: 'many'
25 });
26
27 generator.defineTransition('addRelatedIdea', {
28   semantics: 'link',
29   as: 'relatedIdea',
30   description: 'Add a related idea to this idea',
31   to: 'many'
32 });
33
34 generator.defineTransition('removeRelatedIdea', {
35   semantics: 'unlink',
36   as: 'relatedIdea',
37   description: 'Remove a related idea of this idea',
38   to: 'many'
39 });
40
41 generator.defineTransition('addIdea', {
42   semantics: 'link',
43   to: 'one',
44   description: 'Add an idea',
45
46   parameters: {
47     description: Joi.string(),
48     title: Joi.string()
49   }
50 });
51
52 generator.defineTransition('changeTitle', {
53   semantics: 'modify',
54   to: 'one',
55   description: 'Modify the title of the entity',
56
57   parameters: {
58     title: Joi.string()
59   }
```

```
});

generator.defineTransition('getTitle', {
  semantics: 'link',
  to: 'one',
  description: 'Get the title of an entity'
});

generator.addStartState({
  transitions: [
    { to: 'ideas', via: 'listIdeas' }
  ]
});

generator.addState('idea', {
  type: 'entity',
  parameterized: true,

  attributes: {
    description: { type: 'string', required: true },
    title: { type: 'string', required: true }
  },

  transitions: [
    { to: 'idea', via: 'relatedIdea' },
    { to: 'idea', via: 'addRelatedIdea' },
    { to: 'idea', via: 'removeRelatedIdea' },
    { to: 'idea', via: 'changeTitle' },
    { to: 'title', via: 'getTitle' }
  ]
});

generator.addState('ideas', {
  type: 'repository',

  transitions: [
    { to: 'idea', via: 'addIdea' },
    { to: 'idea', via: 'showDetail' }
  ]
});

generator.addState('title', {
  type: 'service',
  verb: 'get',

  action: function (req, res) {
    var entity = req.params('entity');
    res.json({ title: entity.get('title') });
  }
});

generator.generate();
```

Listing 7.1: The according Foxx Generator

FoxxGenerator will generate a repository and the according collection for ideas. It will create an idea model with the provided attributes. FoxxGenerator will also add an edge collection and the according configuration for ArangoDB to connect two ideas as being related using this edge collection. It will create the following routes:

- `GET /ideaapp`: The start state with an outgoing link to the list of ideas.
- `GET /ideaapp/ideas`: A list of all ideas with links to each of the individual ideas and an action to create a new idea.
- `POST /ideaapp/idea/{entityId}/links/addRelatedIdea`: Add one or more related ideas to this idea.
- `PATCH /ideaapp/idea/{entityId}`: Change the title of a given idea.
- `GET /ideaapp/idea/{entityId}/getTitle`: Get only the title of the given idea.
- `POST /ideaapp/ideas`: Create a new idea with the given parameters.
- `GET /ideaapp/idea/{id}`: Get the representation of a single idea. This representation contains links to all related ideas of this idea.

7.3 Authentication

To allow the use of authentication in FoxxGenerator with different authentication methods like OAuth 2.0, we used a suite of Foxx applications provided by the ArangoDB community. With the help of this applications we have all necessary endpoints to authenticate with services like GitHub[1], Facebook[2] or Twitter[3] and get identity information from these services. These applications can be installed separately alongside a standard Foxx application or a Foxx application generated with FoxxGenerator.

As none of the standards we looked at describes authentication, the authentication works as they were implemented by these applications. The information about the identity of the user is made available to the user of Foxx Generator. It can for example be used in the transition conditions to keep unauthorized users from both seeing a link and using the endpoint.

When the authentication module has been added to ArangoDB one can use the authentication and session information in FoxxGenerator. One use case for this is controlling access to certain routes. In FoxxGenerator this is done by setting a condition on a transition. This will result in both hiding the outgoing link in the representation of the state the transitions starts from as well as checking the condition in the generated route. Listing 7.2 shows how to check if the user is logged in before allowing to create a new idea.

[1] `https://developer.github.com/v3`
[2] `https://developers.facebook.com/docs/facebook-login/v2.1`
[3] `https://dev.twitter.com/oauth`

```
generator.defineTransition('addIdea', {
  semantics: 'link',
  to: 'one',
  description: 'Add an idea',

  condition: function (req) {
    return !!req.session.get('uid');
  },

  parameters: {
    description: Joi.string(),
    title: Joi.string()
  }
});
```

Listing 7.2: Example for using authentication in FoxxGenerator

7.4 Background Tasks with Retry Functionality

By the time the thesis started, Foxx did not support communicating with external services without blocking during the request-response-cycle. In order to address this requirement we implemented Foxx.Queue together with people from the ArangoDB team. ArangoDB has a request queue for incoming HTTP requests with JavaScript workers to answer each of them. These requests are not persistent: If the database shuts down, all remaining items on the queue will be forgotten. This concept has been enhanced to feature a consistent queue backed by an ArangoDB collection. Job descriptions are saved in this collection and the workers then execute these jobs in addition to working on the HTTP requests. An API is available to put jobs onto this persistent queue. If the server shuts down for some reason the jobs will still be stored and can then be executed when the server is restarted.

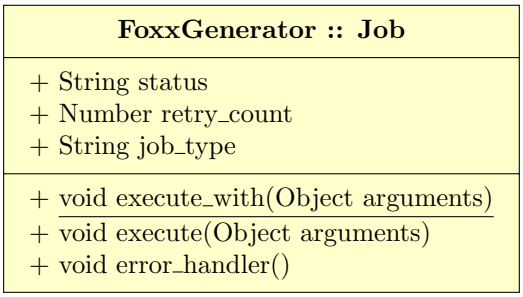

Figure 7.3: UML class diagram for next version of Foxx: Job

This functionality is wrapped in a class of the Foxx framework called `Foxx.Job` which is described in figure 7.3. A job description has the following attributes:

- `status`: Has the job been worked on yet or has it failed?

- `queue`: In which worker queue should this job be worked on?
- `type`: The type is used to identify which JavaScript function should work on this task.
- `failures`: If the job failed, this can be used to store details (for example stack traces) of the failure.
- `data`: The parameters of the request.
- `backOff`: Between each retry the workers will wait for a specified time.
- `maxFailures`: When a job failed, it will be retried for a configurable number of times.
- Handlers for success and failure. As described above, they can not influence the response of the request, but the functions can be used for logging etc.
- Created and Modified timestamps.

In listing 7.3 we first register a new job type that prints the submitted data to the console and then pushes a new task of this kind every time a certain route is visited. With the help of Foxx.Queue multiple service adapters have been implemented: SendGrid[4], Postmark[5], postage[6], mailgun[7], Bugsnag[8] and Segment[9].

```
1  var Foxx = require("org/arangodb/foxx");
2  var ctrl = new Foxx.Controller(applicationContext);
3  var queue = Foxx.queues.create("my-queue");
4
5  Foxx.queues.registerJobType("log", function (data) {
6        print(data);
7  });
8
9  ctrl.get("/log", function () {
10       queue.push("log", "Hello World!");
11 });
```

Listing 7.3: Example for a Job with Foxx

The asynchronous service states use the queue to do the work outside of the request-response cycle. They both define the job type as well as generate the code necessary to put the job description on the queue when the state is visited.

To use the job queue in FoxxGenerator, the developer has to use a `asyncService` state. In listing 7.4 we create a state that calculates the fibonacci number at a certain

[4] https://sendgrid.com
[5] https://postmarkapp.com
[6] http://postageapp.com
[7] http://www.mailgun.com
[8] https://bugsnag.com
[9] https://segment.com

position. This is defined in the `action` attribute of the state. Furthermore we define handlers for success and failure of the calculation. The job is only allowed to fail once and all jobs will be put on a queue named `fibonaccis`. A transition leading to this state needs to have a parameter `calculateFor` in order to work – as this is used in the action function.

```
1  generator.addState('fibonacci', {
2    type: 'asyncService',
3
4    action: function (data) {
5      var calculateFor = data.calculateFor;
6
7      var a=0, b=1, i, temp;
8
9      for (i = 0; i < calculateFor; i++) {
10       temp = a + b;
11       a = b;
12       b = temp;
13     }
14   },
15
16   success: function () {
17     require('console').log('Calculated result');
18   },
19
20   failure: function () {
21     require('console').log('Could not calculate');
22   },
23
24   maxFailures: 1,
25
26   queue: 'fibonaccis'
27 });
```

Listing 7.4: Example for an asynchronous service in FoxxGenerator

7.5 Guidelines

All resulting code is open source and published under the Apache 2 license. It is implemented in ECMAScript 5.1 [ECMA51] targeting the V8 runtime running inside of ArangoDB. We will refer to it via its commonly used name JavaScript. As JavaScript is missing certain features of a class-based object oriented language, we simulated them using best practices from the JavaScript community:

- We simulated namespaces using JavaScript objects.
- ArangoDB provides an implementation of CommonJS modules[10]. Using modules we can split the code into multiple parts, making it easier to understand.

[10] `http://wiki.commonjs.org/wiki/Modules/1.1`

- As JavaScript is a prototype-based object oriented language, we used prototypes as a stand-in for classes. Static methods are defined to be functions on the prototype. Inheritance is implemented using the `extend` function from the Backbone framework[11]. This has the added benefit of allowing us to define both static and instance methods.
- Private fields and methods are not available in JavaScript. All fields and methods marked as private are implemented as module internal and therefore not available from the outside. If they are protected they are modeled as public.
- Neither abstract classes nor interfaces are available in JavaScript and therefore are not used in our modeling.
- As JavaScript only offers weak dynamic typing, all types are just for documentation.

We used the code style conventions as suggested by Douglas Crockford [Croc08] which have been verified using JSHint[12]. To test the generated APIs, we wrote tests using Cucumber in Ruby that describe the desired API to ensure that the generated code leads to the same web API.

[11] `http://backbonejs.org/docs/backbone.html#section-208`
[12] `http://www.jshint.com`

8 Evaluation

In our evaluation we planned to answer the following questions:

- **Speed of Development:** Is the declarative approach outlined in this paper faster than development in a non-declarative web framework?
- **Closeness to Domain:** Is the resulting API easier to understand for a domain expert when it is created using the outlined approach?
- **Resistance to Change:** Does the use of hyperlinks, URL templates and forms increase the resistance to change of the resulting API?

8.1 The Evaluation in Detail

For our evaluation we first talked to two domain experts from different domains. We explained the idea and techniques of domain driven design to them and then wrote a description of a small problem in the language of each of the domains together with the domain experts. We asked both domain experts to introduce a change in the requirements in the form of a second description. For the evaluation we then found two test groups of web developers that are not familiar with the domain. We will refer to the groups as **group A** and **group B**. Group A used FoxxGenerator to implement an API, developers in Group B could choose an existing framework they have experience with to do the same. All of the development sessions were run with the think-aloud technique introduced by Nielson et al. [NCYs02].

8.1.1 Development Speed

Each of the developers in group A was introduced to FoxxGenerator and the ideas behind it in a one-to-one session. As the other group was familiar with the tool they used, this is important to make the results comparable. Each of the test subjects implemented an API including documentation to fulfill the requirements from the description the domain expert created. We compared the time required by each of the teams, with each person getting at most 3 hours to complete the task including the change request.

8.1.2 Closeness to the Domain

Subsequently we showed the APIs to our domain experts and asked them to rate the APIs. In particular the following questions were addressed and rated:

- Does the domain expert understand the resulting API?
- Is the documentation helpful and understandable?
- Are all described requirements met?
- Can she use it to fulfill all use cases?
- Would the change request have broken existing clients?

8.1.3 Resistance to Change

It was planned to interview a **group C** to write API clients that consume the produced APIs in two steps: First implement the version before the requirement change and then adapt it to the change request. We wanted to see if the old client would still work with the API after the change. As only two of the seven APIs were finished we decided to skip this part of the evaluation. Instead we discussed the resistance to change with the domain experts. The comparison using API clients will be done in future work.

8.2 The Domain Descriptions

We created two domain descriptions together with the domain experts which are both extracted from projects they worked on or currently work on. The first domain description was created together with Ingo Friepoertner, a domain expert in car leasing. The description was originally in German and has been translated to English. Dirk Breuer, a domain expert for vacation homes, helped us to create the second description.

8.2.1 Ordering company cars for employees

A company wants to provide employees in different management positions with a company car. The company has a framework contract via a leasing company to cover financing, insurance, claims management and fuel card handling. The company offers its employees pre-calculated car configurations which can be augmented with individual configurations.

An employee should be able to order a company car via an User Interface (UI) that should be integrated into the existing smartphone application of the company and the website. For those interfaces (which will be implemented in a different project) you should create the backend services. The website needs authentication with email and password. Every user has attributes to describe if the user is authorized to have a company car and the management level of the employee (implementing registration is not necessary, you can predefine a set of users).

Every employee that is authorized to have a company car belongs to a management level (level 1, level 2 or level 3) and is allowed to choose from the cars and packages that are assigned to the management level.

At the end of the process the driver will receive a breakdown of the configured car and a list of added extras. The car can then be ordered. In the ordering process the

employee needs to provide a shipping address and a car dealership via which the car will be ordered.

After ordering the employee will receive an email as a confirmation. This information will be listed in the confirmation:

- Date
- Car
- Package
- Leasing rate
- Leasing period and leasing mileage
- Car dealership
- Approximated delivery date
- Shipping address

There should be an UI to see the status of the order. The backend service needs to provide car, order code and approximated delivery date.

Management level

- Level 1: Upper management (max 1200 Euro leasing rate per month)
- Level 2: Middle management (max 800 Euro leasing rate per month)
- Level 3: Lower management (max 600 Euro leasing rate per month)

Cars

Leasing period and leasing mileage

The leasing rate is calculated from the price of the car, the leasing period and leasing mileage considering interest rate, risk category and other factors. This functionality is provided by a mocked web API which expects three query parameters (price, kilometers and period) and returns either a JSON object containing the leasing rate or a JSON array of errors. Example call: `http://leasing-rate-calculator.herokuapp.com/?price=60000&period=48&kilometers=80000`

The API will return an error when the leasing period or mileage is not one of the following values:

- Leasing mileage: 10.000 km, 20.000 km, 40.000 km, 50.000 km, 80.000 km, 120.000 km
- Leasing period: 24 months, 36 months, 48 months

Management Levels	Name	Price
3	Audi A3 Sportback S line 1.8 TFSI 6 speed	34.000 Euro
2,3	Audi A4 Avant Black Edition 2.0 TDI multitronic	32.000 Euro
1	Audi Q7 S line Sport Edition 3.0 TDI quattro tiptronic	70.000 Euro
1	Audi S8 4.0 TFSI quattro tiptronic	93.000 Euro
2,3	BMW 220i Coupé Steptronic	32.000 Euro
1,2,3	BMW 328i Touring	42.000 Euro
1,2,3	BMW X5 sDrive25d	55.000 Euro
2,3	VW Golf Variant TDI BlueMotion Comfortline	23.700 Euro
1,2,3	VW Passat Trendline 2.0 TDI BlueMotion	31.300 Euro
1,2,3	Mercedes E 220 BlueTEC Limousine	44.120 Euro
1	Mercedes E 500 4MATIC Limousine	73.631 Euro
1	Mercedes S 300 BlueTEC HYBRID Limousine	80.920 Euro

Table 8.1: Electable cars with management level and price

Packages

Level 1 and 2 can choose from both packages, level 3 always includes package 1. The employee can only choose one package.

- Package 1: Claims management plus, Pick-up service, rental service, free choice of workshop and fuel card
- Package 2: Claims management plus, Pick-up service, partner workshop and fuel card

Car dealerships

- Autohaus Adam
- Autohaus Moll
- Autohaus Peters
- Huddel & Brassel KG
- Heitmeier GmbH & Co. KG

Change Request

The employee should be able to select the following extras depending on the car:

- Entertainment System V3 (Audi, not A3): 200 Euro
- Apple – CarPlay (Mercedes, BMW, Audi): 1000 Euro

- Open Automative Alliance – Automotive Link (Audi, GM, Honda and Hyundai): 500 Euro
- Coupling device (all cars): 30 Euro
- Isofix Bracket (all cars): 10 Euro

The extras can be added up to the maximum leasing rate. Exceeding the rate leads to co-payment which has to be shown for the order.

8.2.2 Searching for vacation homes in Denmark

The result of the task has to be an API which can power a website that implements the following specification.

Description

As a user I want to be able to search for vacation homes in Denmark by specifying multiple search criteria. The criteria must include the area of the house, the maximum price, the arrival date (only Saturday is allowed as arrival date), the duration of the stay (in weeks with a maximum of four weeks), the number of people traveling and a list of optional extras a house may have (Internet access, dish washer, sauna, fireside, sea view).

To select an area of interest I want to see a map of Denmark on which the available areas are clickable. Additionally I want to select multiple areas to be added as search criteria.

As a user I want to be able to select each of the above mentioned criteria in a web form. After configuring the criteria I want to start the search by pressing a red button. Submitting the form should result in an ordered list of houses matching the entered criteria. The default order of the list should be the price of the houses in ascending order.

On the result page I want to be able to further add, remove or change all of the available criteria. The list of results should be updated accordingly.

Acceptance criteria

- When submitting the form without any criteria I expect to see a list of all available houses in the system.
- When submitting the form with any criteria I expect to see a list with only houses matching the specified criteria.
- When on the result page I want to see the same search form as the one on the start page with the currently active criteria visible.
- When on the result page I want to change the criteria in the form and the list of houses will change accordingly without a full page reload.

- When navigating from the result page to a single house page I want to see a link that brings me back to the search result page.
- When the specified search criteria don't result in any matching houses, an appropriate message should be shown to the user.

Note to implementers

- It is not expected to implement an actual search but rather return a fixed list of IDs to houses
- The following list of areas can be used:
  - Aerö
  - Albaek
  - Als
  - Argab
  - Bjerregard
  - Blavand
  - Blokhus
  - Bork
  - Bornholm
  - Bratten
  - Ebeltoft
  - Falster
  - Fanö
  - Faxe
  - Fejø
  - Fjand
  - Fjellerup
  - Gjellerodde
  - Hejls
  - Henne

Change Request

As a user I want to additionally search for houses that belong to a specific partner. A partner within the system is the original owner of a house. The customer will conclude a contract with that partner and **not** the platform providing the search.

Participant	Task	Tool
LF1	Leasing	FoxxGenerator
LF2	Leasing	FoxxGenerator
LN	Leasing	Node.js
LR	Leasing	Rails
VF1	Vacation Homes	FoxxGenerator
VF2	Vacation Homes	FoxxGenerator
VP	Vacation Homes	Padrino

Table 8.2: Participants of the evaluation and the tools they used

A partner can own any number of houses. A partner itself is just defined by its name. Additionally a partner can be either active or inactive. Inactive partners are present in the system, can own houses and will appear in the backoffice, but their houses are not available to the end user.

As a user I want only to see a list of active partners.

Acceptance Criteria: On the search form there should be a select box with a list of all active partners. Selecting one or more partners will limit the search result to houses that are owned by the specific partners.

Example Data for partner list: Sonne und Strand, DanCenter, Novasol, Dansommer, Feriepartner Bork Havn, Feriepartner Rømø, NetFerie, Fejø Feriehuse, Feriepartner Thy, Feriepartner Jammerbugten and Admiral Strand.

8.3 Results from the think aloud sessions

In order to keep the anonymity of the participants of our evaluation we used a naming schema. In table 8.2 we listed the participants using this naming schema with the task they were supposed to solve and the tool they used.

8.3.1 LF1 using FoxxGenerator to solve the leasing problem

LF1 started with drawing the statechart after reading the description. The statechart as shown in figure 8.1 (replica of the drawn original) was finished after 20 minutes of drawing: It is supposed to show the flow as it is described in the task. The participant decided to follow three different paths for the three management levels.

LF1 could not finish the task with the prototype as it was not possible to capture the current state in the statechart: How does one save which car, extras etc. have been chosen? The participant tried to save the information about that in the session data for the current user, but could not finish it in time. The design would have lead to encoding the state outside of the statechart.

The following issues where noted during the think aloud session:

- It is not possible to filter the contents of a repository. Filtering the cars or packages was therefore not possible.

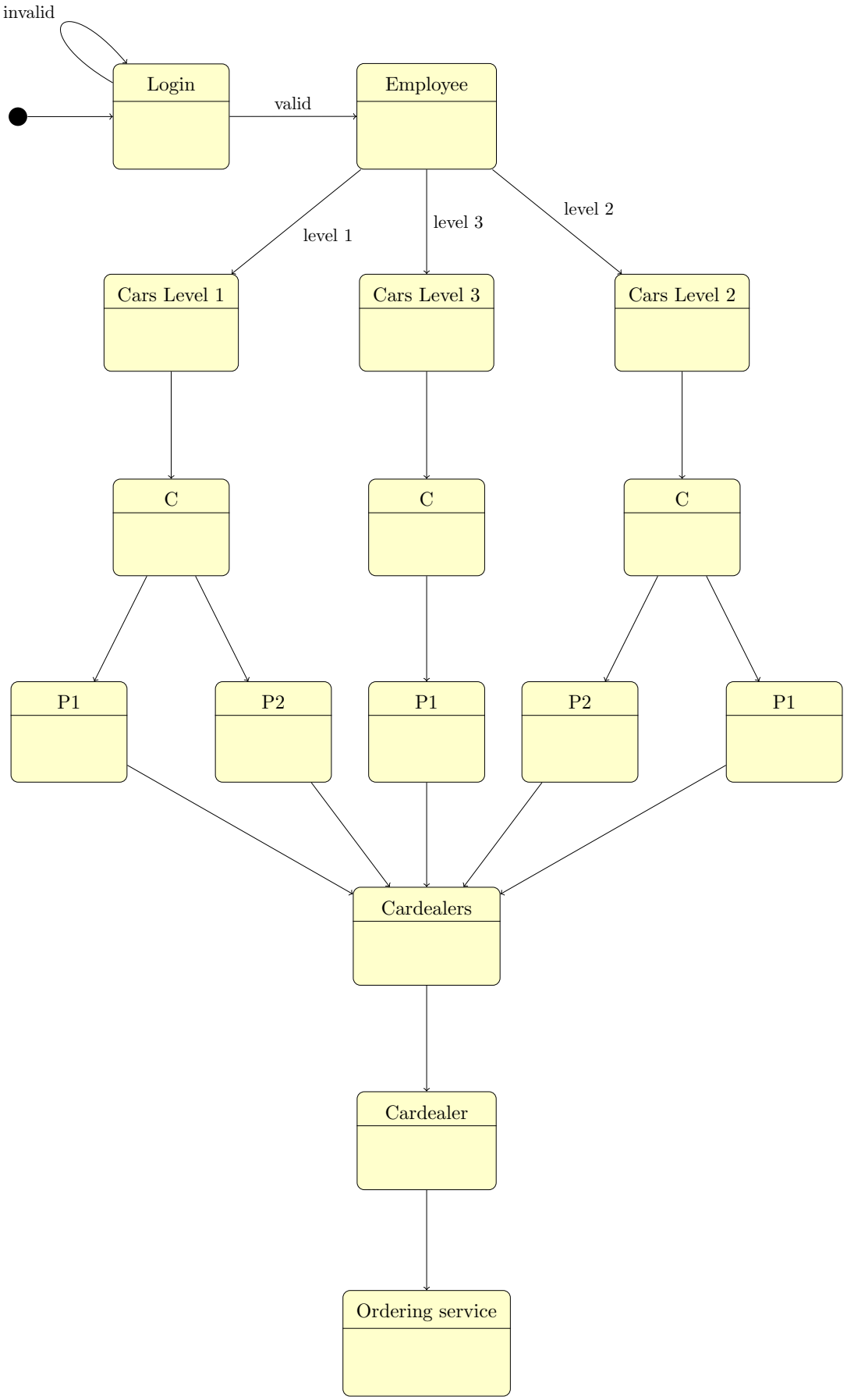

Figure 8.1: Replica of the statechart as drawn by LF1 to solve the leasing problem

- LF1 expected a service to rather be a kind of transition.
- It is required to have a `link` transition between a repository and an entity in order for saving to work. This is counterintuitive and creating entities was not necessary in the given description. LF1 worked around it by setting the condition to always return false therefore hiding the link and making the route not available.
- It needs to be possible to define a `reverseTransition` for a transition between a state and a parameterized state which leads back to the state with the according parameters.

The domain expert found that the resulting API would have fulfilled his expectations. The result was a little surprising to him when compared to the APIs he already knew, but he liked the graphical representation as a foundation for discussions. Compared to the APIs built with the other two tools he could see a workflow in the description reflecting his description. Therefore it was easier for him to compare it to his description. The workflow chosen by LF1 was too rigid for him – especially compared to the solution by LF2. He also did not like that he had to go back to the beginning of the statechart in the case of a negative result at the end.

8.3.2 LF2 using FoxxGenerator to solve the leasing problem

LF2 used OmniGraffle[1] to create the statechart shown in figure 8.2. The participant said that in the design the main focus was on the workflow which is in the task description. During the design process LF2 multiple times asked himself where the user is in the design as the participant could not find a good spot for a user state. LF2 then decided to not make the user part of the statechart. When designing, every state and every transition got a name at a very early stage. The types of both where determined at a very late point in the design. It also happened multiple times that LF2 converted states into transitions as the participant found them to be a better fit for what needed to be explain. This happened for example when the participant wanted to check if the order is complete. This was first implemented as a service that would check the entity and was then changed to a transition with a condition. As the condition would prevent the link to the next step of the process to be shown, the user of the API would be able to check if the next step is possible now.

LF2 further noted that a generator concept in combination with a REST standard would help decreasing the number of discussions about details.

When discussing the statechart we realized that the prototype would not be able to generate the according API for the statechart as features are missing. Instead LF2 marked the areas where the statechart would need to change in order to to fulfill the changing requirement (they are marked bold in the figure). LF2 did not run into the same problem as LF1 as the participant saved all information about the selection in an

[1] `https://www.omnigroup.com/omnigraffle`

entity. LF2 however would have required a transition that creates an entity that did not start at the repository. It would also have been necessary to take the information about which selection has been created (e.g. its ID) through multiple service states. For the statechart it would also have been necessary to have a combination of a repository and entity in one state, but this is not possible with the generator. LF2 further noticed the following shortcomings of the prototype:

- There should be the possibility to introduce ϵ-edges between states to indicate that the transition should be done automatically. This is especially the case for leaving services after they are done with their action.
- The transition to a service should have its own semantics called *execute*.
- LF2 found it to be problematic that the output of a state is not visible in the statechart. Changing the output can lead to breaking changes in the client and should be signalized in the statechart.
- It should be possible to denote a transition between two entities as the first entity containing the second entity (which should be reflected in the URL as one being a subresource of the other).

As already described in section 8.3.1 the domain expert was surprised about the representation, but liked it for discussing about it – and again appreciated seeing the workflow he had described. He really liked the resulting workflow that has a lot of flexibility in the order of taking the steps. By using the conditions it is still ensured that if a step needs to be taken before another one that this is fulfilled. The domain expert also liked the reasonable back transitions to avoid the need of going back to the start state. Seeing the statechart also raised questions that he would have discussed with the developer to clarify his description. Some of these things could not have been annotated in the statechart, but could still could have been implemented in the generator DSL.

8.3.3 LN using Node.js to solve the leasing problem

LN chose Node.js using the Express web framework[2] to create the API. LN also added a few more libraries during the design from NPM. While reading the description from the domain expert the participant analyzed what kind of data the API would need to provide: Cars, users, packages and car dealerships. Then LN started writing the code for the implementation.

LN first created JSON files for cars, users, packages and car dealerships. The participant would have use a database in that case, but as the data was static for now and the time was limited the participant would just use JSON files. For cars and suppliers LN created a simple API endpoint that would just deliver the JSON file. LN briefly

[2] `http://expressjs.com`

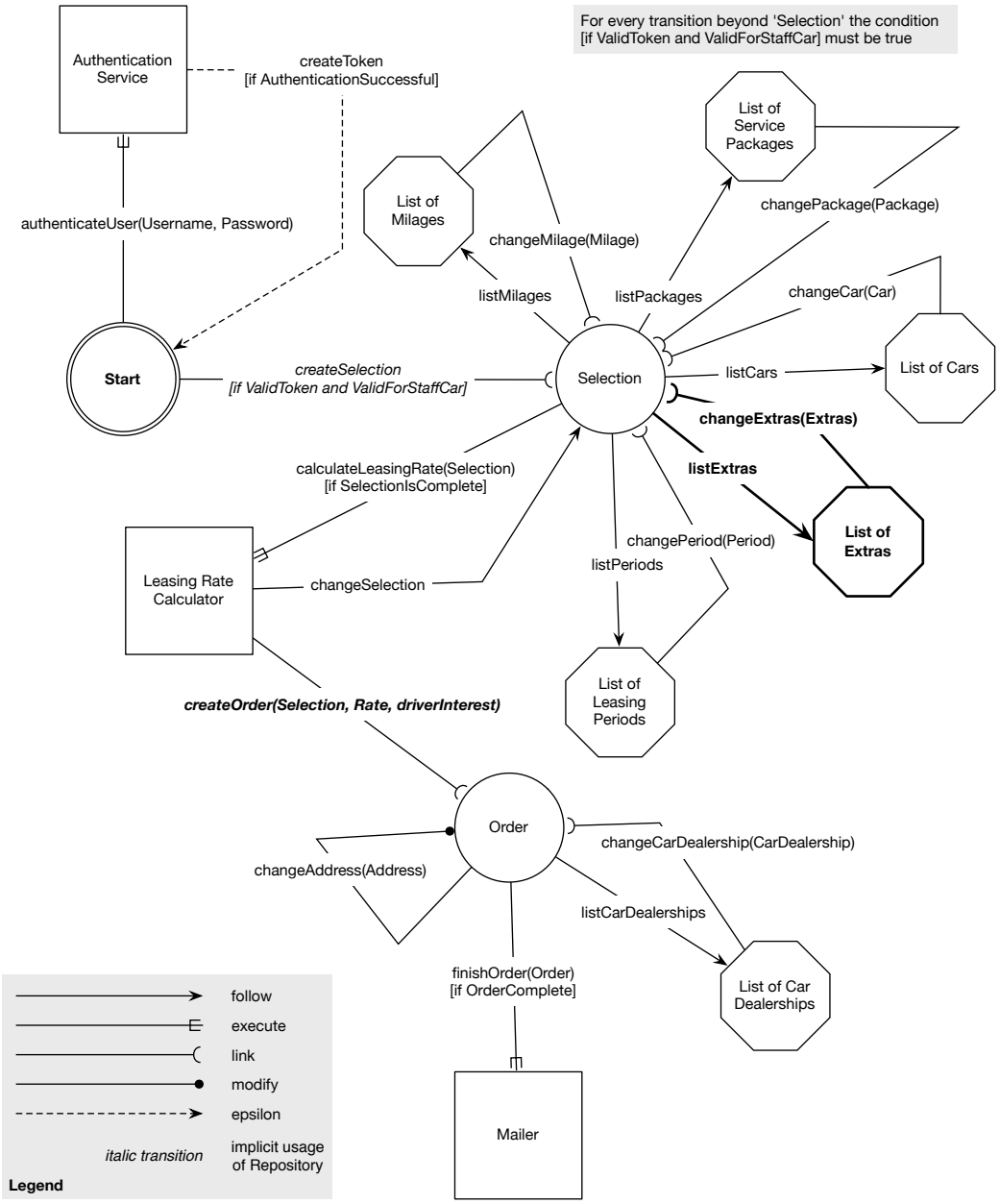

Figure 8.2: Statechart as drawn by LF2 to solve the leasing problem

checked if they return the expected value using cURL[3]. As the participant wanted to filter the list of cars depending on the user level a simple cookie authentication was implemented.

At this point LN noted that cURL would no longer work for him to try out the API as it was too much hassle to handle cookie authentication with this tool. LN therefore continued working without checking if the result was as expected. After implementing an endpoint for receiving the filtered list of cars LN created an endpoint to validate an order. This endpoint expects an order object containing all information necessary to create an order and returns if this order would be valid by first checking if all arguments are present and then communicating with the external service to check for the leasing rate. Then LN created a second endpoint that expected the same arguments plus the address and car dealership which would first do the same validation and then save the order and returned an UUID. Finally LN added an endpoint to get the status of an order by its UUID.

To check if the API was working as LN expected an HTML site that included jQuery was created which was delivered by the Node.js application and LN used the JavaScript console to communicate with the API. This way the browser would save the cookie and send it to the server alongside every request. After checking that everything worked as LN expected the participant declared the API done after 2 hours of work.

In order to fulfill the change request LN accepted an additional argument for extras at both the validation endpoint as well as the order endpoint. The extras default to an empty array to keep backwards compatibility with old clients. Changing the requirement of allowing the user to go over the limit but paying it themselves could not been implemented without breaking backwards compatibility. Changing the requirement took 30 minutes.

We asked him what LN would have done differently given a bigger time frame:

- LN would have used a database instead of JSON files.
- LN would have written a documentation for each of the endpoints using Markdown. In this documentation LN would have added a description of the parameters for each of the POST routes.
- LN would have written tests using a Node.js library to test the behavior.

The final version had the following routes:

- `GET /`: Deliver the blank page for testing.
- `POST /auth`: Login with email and password.
- `GET /whoami`: Get information about the currently logged in user (for checking if auth works).
- `GET /cars`: Get the list of cars filtered for the management level of the user.

[3] `http://curl.haxx.se`

- `GET /suppliers`: Get the list of car dealerships.
- `GET /services`: Get the list of packages.
- `POST /validate`: Validate the parameters to create an order.
- `GET /orders`: Get a list of all orders created by the user.
- `GET /orders/:id`: Get information about a specific order.
- `POST /orders`: Create a new order.

The domain expert found that the API fulfills all requirements. He did not like some of the error responses as one error would overwrite other errors in the response. Due to only one endpoint being responsible for creating the entire order it was a challenge for him to check if all his requirements were fulfilled. This could only be done by trying out the endpoints with different requests using jQuery.

8.3.4 LR using Rails to solve the leasing problem

LR chose Ruby on Rails[4] to create the API using PostgreSQL[5] for persistence. After reading the description LR made notes about the resources found in the description: A resource in Rails is a combination of a database table, the according representation in Ruby and the API endpoint. Then LR noted use cases of the API which the participant would have written as acceptance tests with more time to complete the task.

LR created the API using test driven development with RSpec[6]. When developing LR made extensive use of the generator functionality of Rails to create the resources LR noted down earlier. In order to provide authentication, LR first wanted to implement a token based authentication. Due to the time restrictions LR decided to use basic authentication instead.

At the end of the three hours LR created the following routes for the API:

- `GET /cars`: Receive a list of cars (it was planned to filter the list, but did not fit into the given time frame).
- `POST /car_configs`: Create a car configuration using all parameters from the description and the external service. The configuration is saved to the database.
- `POST /car_configs/:car_config_id/order`: This route was supposed to create the order from the configuration. It was not finished in time.
- `GET /orders/:id`: This route was prepared, but not finished. It was supposed to get an order by its ID to receive the status.

[4] `http://rubyonrails.org`
[5] `http://www.postgresql.org`
[6] `http://rspec.info`

The domain expert found that the API fulfills all requirements. He liked the error handling of the API. As the workflow was not visible for him, he used the tests written by LN to check if all his requirements have been addressed. Compared to the solution by LN he evaluated it to be a little more flexible as it allows to order at a later time than configuring the car.

8.3.5 VF1 using FoxxGenerator to solve the vacation home problem

VF1 used FoxxGenerator to create an API for the vacation home problem. While reading the text VF1 wrote down the entities and value objects in the description and how they are connected. Then VF1 drew the statechart in 30 minutes with two iterations, the final iteration is shown in figure 8.3. VF1 then implemented the drawing in the DSL in 90 minutes. VF1 had little experience with JavaScript beforehand, but had no problem using the DSL. After the first few states VF1 started using the interactive documentation to check if the generated API was as expected. VF1 also used the interactive documentation to create the entities that were described by the domain expert like the regions. VF1 described the transition from the drawn diagram to the DSL as natural.

During the think aloud session VF1 made the following suggestions to improve FoxxGenerator:

- FoxxGenerator should print warnings for all states that are not reachable.
- VF1 first wanted to add the functionality of executing the search to be part of the repository, but as this was not possible, VF1 used a service instead.
- It would be nice to be able to just take any state as the start state of the statechart.

After about two hours VF1 had generated the following routes for the API:

- `GET /search`: Billboard URL.
- `GET /search/SearchCriteria`: Get information about the current search.
- `GET /search/Areas`: Get a list of all areas.
- `GET /search/Options`: Get a list of all options.
- `GET /search/Area/:id`: Get information about a specific area.
- `GET /search/Option/:id`: Get information about a specific option.
- `POST /search/SearchCriteria`: Create a new search where each of the parameters is optional.
- `POST /search/Areas`: Create a new area.

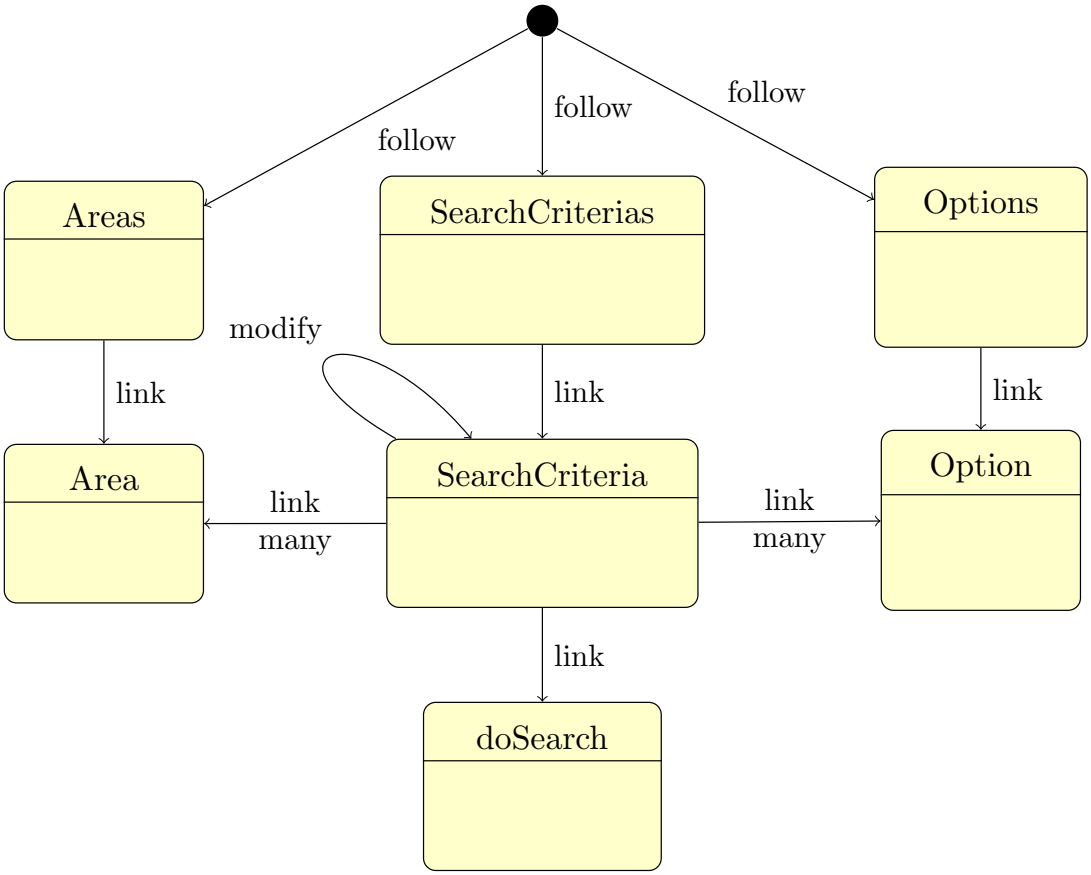

Figure 8.3: Replica of the statechart as drawn by VF1 to solve the vacation home problem

- `POST /search/Options`: Create a new option.
- `POST /search/SearchCriteria/:entityId/links/addOptions`: Add an option to the search.
- `POST /search/SearchCriteria/:entityId/links/addAreas`: Add an area to the search.
- `PATCH /search/SearchCriteria/:entityId`: Change search parameters of a search.
- `GET /search/SearchCriteria/:entityId/doSearch`: Get the results for the specified search.

The domain expert determined that the API fulfills all requirements he formulated. Interactive documentation is a bonus when either pairing or when reviewing the API. Also when debugging or developing a frontend for it. The statechart helps a lot when talking about the API with the developer. The resulting API can also be used by other clients that might be added in the future (mobile apps etc.). He liked the distinction between entities and value objects and easy to follow. It also allows to easily add new areas or options as they are creatable using the endpoints. The resistance to change is very high, as adding another option either adds a value object or another entity that can be linked.

8.3.6 VF2 using FoxxGenerator to solve the vacation home problem

VF2 used FoxxGenerator to create an API for the vacation home problem. After multiple iterations VF2 arrived at the statechart that has been replicated in figure 8.4. Compared to VF1 VF2 decided that the options should be value objects and not entities and therefore did not create a repository for them. The resulting statechart could not be expressed using the prototype as the service would have needed access to the repository state. VF2 suggested to introduce a new relation semantic called `access` which would allow a service to access the data in a repository.

VF2 also had the following ideas during the think aloud session:

- VF2 suggested inheritance between states which VF2 would also have expressed with an edge with the `extend` semantic.
- VF2 multiple times used the expression “adding a service to a repository”. VF2 suggested to handle services as extensions of states rather than its own kind of state.
- VF2 suggested to make it possible to have substatecharts that contain one workflow that is independent of other workflows in the statechart.

The domain expert determined that the API would have fulfilled the requirements. Even though the search was not implemented as an entity, it is still ensured that the

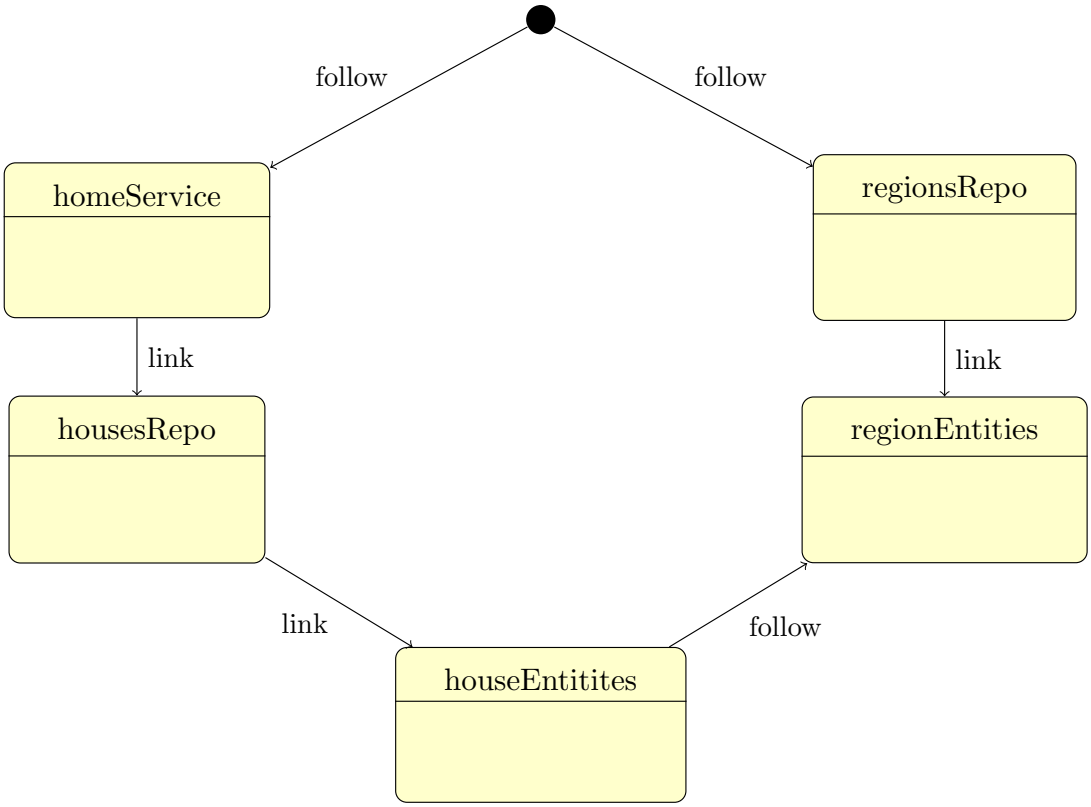

Figure 8.4: Replica of the statechart as drawn by VF2 to solve the vacation home problem

input was validated. He did not like that the extras were modeled as value objects instead of entities. The domain expert was also a little confused why the image displayed things about the data model that he would not have expected in the drawing. We could look at the modeling even though we did not have any implementation. Modeling beforehand leads to finding problems with either the model or the tooling at a very early point of the process. This would have allowed to adjust the requirements. The graphic is an artifact that is valid for quite some time and can be used to explain the modeling and be used for future discussions.

8.3.7 VP using Padrino to solve the vacation home problem

VP used Ruby and the Padrino[7] web framework to solve the vacation home problem. For serializing and deserializing VP used Roar[8]. In the design VP started from the form in the webpage and therefore planned only one endpoint from the beginning. This endpoint decodes all data from the client and then returns the search results. When thinking about how to model the JSON response of the API, VP did a short research on how different APIs did that. All arguments of the search have been treated as value objects. Therefore they were treated as free text were users could put in arbitrary information. Validation of inputs was not implemented.

Overall the process was rather unstructured with a lot of trial and error. Tests were written alongside the implementation using RSpec[9] to check if the API works as expected. The API could not be finished in the given time frame.

The domain expert said that it is acceptable for him to save the state on the client. It is resistant to change. VF2 found it problematic to model all value objects as free texts as they are hard to validate. In the same way there is no endpoint for getting the valid endpoints from the API. He did not see it as a major problem to only have one endpoint, but it might be a problem for future extension. The API is neither self-describing nor documented.

8.4 Comparison and Summary

The domain expert for car leasing evaluated all four APIs as acceptable solutions. He preferred the solution by LF2 to the other ones. He had no preferences when comparing the other solutions. The graphical solution helps a lot for the first draft of the process. All details can be filled in later. Logical bugs have to be obvious, but he was not sure if they would always be visible in the graphical solution. But the graphical solution raised more questions. Discussions can be held earlier, because there is a common ground in the graphical description. While discussing with the domain experts about the statechart several questions were raised that were made visible by the graphical representation. Therewith it helped to identify hidden requirements that

[7] `http://www.padrinorb.com`
[8] `https://github.com/apotonick/roar`
[9] `http://rspec.info`

were unconsciously not mentioned by the domain expert in the textual description. The first statechart was more rigid, you have to go through a certain flow. You have to choose the car before everything else etc. Compared to that, there is no workflow visible in the other two APIs. This has to be discussed and done in a discussion about the frontend.

The domain expert for vacation homes evaluated all three APIs as acceptable solutions. The solution by VF1 was by far easiest to understand and has trust in the resistance to change for the future. From the modeling he got new ideas like saved searches. This was only obvious in this case. He really liked that the description of the API generates a documentation and something to try out things in the API. He would have ranked them: VF1, VF2, VP. The mapping between the drawn model and the implementation is easy (compared to ER diagrams for example) and the tool encourages people to draw this picture. He imagines that if he would have worked with one of the developers using FoxxGenerator, the graphical representation would have been something that can be used for discussions, especially compared to the abstract code or even acceptance tests.

The domain expert for car leasing said that it could be harder to reason about a statechart for a more extensive problem. As a solution he suggested to break the statechart down into multiple smaller charts that could then be connected. He also wished for a possibility to annotate which information would be available at which point in the statechart. Both domain experts said that the statechart is a very effective way of communicating between the domain expert and the developer. It was also very useful to them to find parts of their description that they should clarify. They also both liked that FoxxGenerator forced the developers to validate all inputs.

None of the APIs that was developed using other tools than our prototype used a standard. The developers did not mentioned standards when asked what they would have done differently with more time to solve the task. Each of the developers using the prototype chose Siren over JSON+API as they saw it as a better fit to the way that the described design process results in a workflow.

The four developers using FoxxGenerator to solve their task all liked to model an API using a statechart. This lead to a very structured process to create the APIs. The interactive documentation to try out the generated API was appreciated and intensively used by VF1. Three of the four statecharts could not be used to generate the API as features were missing. In order to generate Web APIs for all statecharts as drawn by the authors we have to implement the following features:

- Persisting a new entity should be possible when transitioning from one state to another and not only if the starting state is the accompanying repository. In some cases of the modeled statecharts the repository state is never visited as an endpoint to receive a list of all entities stored in one repository is not necessary.
- Services need a way to access the data stored in one or more repositories. This could be solved by introducing an `access` semantic.
- Repositories need to be filterable.

- When defining a transition it has to be possible to express that the target state needs information about previous states. This can be used to store information about previously visited parameterized states in the URL. This is especially important to model workflows.
- A `reverseTransition` needs to be possible between a parameterized state and an arbitrary state to allow back links.

From our evaluation we can make the following conclusions:

- Using the outlined design process and the drawn statechart it is easier to communicate with the domain expert. In some cases it even lead to additional explanations from the domain expert that had not been stated clearly in the description.
- The statechart simplified the evaluation of the API by the domain expert.
- Some of the hand written APIs did not return reasonable status codes due to the time restrictions. The generator can deduce the status codes from the semantic description – therefore the developer does not need to specify them.
- None of the hand written APIs had any documentation due to the timing restriction. The developers said that they would have written this documentation by hand afterwards if they had enough time. The generator has enough information to generate an interactive documentation and the resulting API is partially self-documenting.
- FoxxGenerator forces the developers to think about validation. The other tools offered to add validation, but did not enforce it.
- The generated APIs had more API endpoints than the hand written ones.

9 Conclusion and Outlook

In this thesis we introduced a declarative approach that is based upon a combination of the design approach by Richardson and Amundsen [RiAm13], domain driven design [Evan03] and the statecharts by Harel [Hare87]. By modeling with a statechart the result is a Web API that offers links between the endpoints that resemble a workflow. In our evaluation all developers which used this way of modeling found it to be a good process to design a Web API.

We implemented a declarative JavaScript framework that allows the developer to convert a statechart designed with this process into a fully functioning API with persistence including an interactive API documentation. The resulting Web API follows best practices of RESTful API design and follows existing standards. It currently supports two standards: JSON+API and Siren. The resulting API can both use authentication with basic auth or OAuth 2.0 and background jobs.

9.1 Conclusion

In our evaluation we found that using the outlined design process and the drawn statechart made it easier to communicate with a domain expert as the statechart and therefore the resulting API is close to the workflow as described in the domain language. In some cases it even helped to reveal issues the domain expert had not clearly specified in the description. The generated APIs follow best practices of Web API design while at the same time lightening the developers workload. This specifically encompasses:

- The API's return status codes fit the semantic description of the API.
- Explicitly modeling the value objects of both transitions and entities is enforced. The generator uses the modeling to implement validation of all inputs of the API.
- Using the statechart, an interactive API documentation is generated which describes the same semantics as the implementation. When the mental model in the form of the statechart changes, the implementation and the documentation change alongside each other.
- The generated APIs follow a standard. In the case of our evaluation the chosen standard was Siren, which does not have any specific application semantics.

The framework we created is significantly different from existing frameworks to create Web APIs. It always generates an API where the endpoints are linked to

each other. This is both encouraged by the design process as well as enforced by the tooling as resources that are not linked will not have an endpoint. In other approaches developers have to specifically add links between resources.

It was not possible to create a Web API from all statecharts as certain features were missing. The introduced design process however was applicable to both tasks and helped both in designing the Web API as well as communicating with the domain expert. The developers using the design process finished early. The result was already useful to the domain expert. One statechart was drawn and used to generated the Web API in about two hours while the same task could not be completed in three hours by another developer using a tool of his choice.

9.2 Outlook

The current version of the prototype was not able to generate a Web API for each of the statecharts. In order to work with each of them some additional features have to be added. The main problem was that when defining a transition it has to be possible to express that the target state needs information about previous states. This can be used to store information about previously visited parameterized states by putting the parameter of the state in the URL. As this is especially important to model workflows this will be implemented first. It is also necessary to extend the functionality of both services and repositories.

Currently the process of designing the statechart has to be done either with pen and paper or with graphics software. It then has to be translated into the DSL introduced in this thesis manually. Even though the process was straight forward for the developers in the evaluation it would still be easier to use a graphical editor that will immediately generate the Web API. This would lead to an even closer feedback loop than with the current prototype.

Bibliography

[Stro98] Carlo Strozzi. NoSQL – A Relational Database Management System.

[Croc08] Douglas Crockford. *JavaScript: The Good Parts: Unearthing the Excellence in JavaScript*. O'Reilly Media / Yahoo Press, 2008.

[ECMA51] Ecma International. ECMAScript® Language Specification, 2011.

[Evan03] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison Wesley, 1. a. edition, 2003.

[Fiel00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, Irvine, 2000.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Hack13] Michael Hackstein. Graph-Based GUI for the NoSQL-Database ArangoDB. Master's thesis, RWTH Aachen University, 2013.

[Hare87] David Harel. Statecharts: A Visual Formalism For Complex Systems. Science of Computer Programming, pages 231–274. North-Holland, 1987.

[json14] Internet Engineering Task Force. The JavaScript Object Notation (JSON) Data Interchange Format, 2014.

[JSR339] Java Community Process. JAX-RS 2.0: The Java API for RESTful Web Services, 2013.

[Klab13] Steve Klabnik. *Designing Hypermedia APIs*. Self Published, 2013.

[LSSc12] Olga Liskin, Leif Singer, and Kurt Schneider. Welcome to the Real World: A Notation for Modeling REST Services. *Internet Computing, IEEE*, 16(4):36–44, 2012.

[Dohm12] Lucas Dohmen. Algorithms for Large Networks in the NoSQL Database ArangoDB. Bachelor's thesis, RWTH Aachen, Aachen, 2012.

[Kell13] M. Kelly. JSON Hypertext Application Language, 2013.

[MeGr09] Peter Mell and Tim Grance. The NIST definition of cloud computing. *National Institute of Standards and Technology*, 53(6):50, 2009.

[Amun13] Mike Amundsen. Collection+JSON - Document Format, 2013.

[MiPo13] Michael S. Mikowski and Josh C. Powell. *Single Page Web Applications*. Manning Publications Co., 1 edition, 2013.

[MaPD10] Maria Maleshkova, Carlos Pedrinaci, and John Domingue. Investigating Web APIs on the World Wide Web. *The 8th IEEE European Conference on Web Services*, 2010.

[MWDT07] E.Michael Maximilien, Hernan Wilkinson, Nirmit Desai, and Stefan Tai. A Domain-Specific Language for Web APIs and Services Mashups. In BerndJ. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *International Conference on Service Oriented Computing – ICSOC 2007*, volume 4749 of *Lecture Notes in Computer Science*, pages 13–26. Springer Berlin Heidelberg, 2007.

[NCYs02] Janni Nielsen, Torkil Clemmensen, and Carsten Yssing. Getting Access to What Goes on in People's Heads? Reflections on the Think-aloud Technique. In *Proceedings of the Second Nordic Conference on Human-computer Interaction*, NordiCHI '02, pages 101–110, New York and NY and USA, 2002. ACM.

[PaWi09] Cesare Pautasso and Erik Wilde. Why is the Web Loosely Coupled?: A Multi-faceted Metric for Service Design. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 911–920, New York and NY and USA, 2009. ACM.

[RiAm13] Leonard Richardson and Mike Amundsen. *RESTful Web APIs*. O'Reilly, 1 edition, 2013.

[RiRu07] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly, 2007.

[RSKl12] Dominik Renzel, Patrick Schlebusch, and Ralf Klamma. Today's top "RESTful" services and why they are not RESTful. In *Web Information Systems Engineering-WISE 2012*, pages 354–367.

[SaFo12] Pramond J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley, 2012.

[Thou13] Thoughtbot Inc. *thoughtbot Playbook*. Self Published, 2013.

[Tibc14] Tibco Software GmbH. Understanding the New Open API Economy, 2014.

[xml08] W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition), 2008.

[ZuSc12] Ivan Zuzak and Silvia Schreier. ArRESTed Development: Guidelines for Designing REST Frameworks. *IEEE Internet Computing*, 16(4), 2012.

List of Figures

List of Tables