TECHNISCHE UNIVERSITÄT
MÜNCHEN

DEPARTMENT OF INFORMATICS

Master's Thesis in Informatics

A Model-Based Approach to Consume REST Services in Single Page Applications

Niklas Scholz

TECHNISCHE UNIVERSITÄT
MÜNCHEN

DEPARTMENT OF INFORMATICS

Master's Thesis in Informatics

A Model-Based Approach to Consume REST Services in Single Page Applications

Ein Modellbasierter Ansatz für die Nutzung von REST-Services in Einzelseiten-Webanwendungen

Author:	Niklas Scholz
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	M. Sc. Adrian Hernandez-Mendez
Submission Date:	15.10.2017

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 11.10.2017 Niklas Scholz

Acknowledgements

I want to thank everyone who supported me through the process of researching and writing this thesis. Especially, M.Sc. Adrian Hernandez-Mendez for his guidance and valuable feedback. Furthermore, I want to thank M.Sc. Manoj Mahabaleshwar for supporting me during the evaluation of the code generation tool. I would also like to acknowledge Prof. Dr. Matthes for supervising my thesis. Finally, I want to express my very profound gratitude to my family and friends who always encouraged me during my studies and the entire process of this thesis. Thank you!

Abstract

Single Page Applications are often communicating with backend servers to transfer data. To make backend services accessible for the client, the backend provides those via RESTful APIs. A lot of research and code generation tools focus on the server side of such RESTful APIs. Whereas, this thesis deals with the client-side, meaning the consumption of RESTful APIs. Consuming APIs leads to complexity in the client when the data comes from several different APIs. This is a common case when applying the microservice architectural style or when the client accesses several third party APIs (e.g. GitHub, JIRA, Facebook, Google API etc.).
This work presents an approach to shift away the responsibility of the API consumption from the client by introducing a web service which handles the communication with the APIs as well as the data transformations. After analysing the state-of-the-art, a model-driven approach is being proposed for the generation of such a web service. Firstly, a model will be introduced describing the consumption of RESTful services. Secondly, a tool will be described to semi-automatically generate such a web service based on model-transformations. Subsequent to the presentation of the implementation, the approach will be evaluated by developing a sample application with the help of the tool. The outcomes of this thesis are a reference architecture, a meta-model describing RESTful service consumption, and a code generation tool. The conducted research leads to the conclusion that consuming several RESTful API leads to complexity in the client and that this consumption can be modelled. Furthermore, evaluating the approach implies that this complexity can be reduced with the presented approach.

Contents

List of Figures

List of Tables

Listings

1 Introduction

If you're afraid to change something it is clearly poorly designed.

Martin Fowler

Contents

1.1 Motivation and Problem Description

Nowadays web sites are powerful applications where several computer systems communicate with each other on the internet. In single page applications this communication often is not just the traditional three-trier connection between the frontend, a backend and a server as it used to be. It is more and more common that the application frontend communicates with several different servers. Especially with current software hypes and trends such as microservices or serverless architectures, clients access multiple RESTful APIs. In a serverless architecture, for example, clients connect to third party applications which provide ready-to-use functionality. Those can be services supplying cloud database access, authentication or simply data [1]. However, also the rising popularity of microservices changed the traditional client-server communication. In a microservice architecture web clients have to handle communication with 20 or 30 RESTful services. Particularly, when an API was designed by a third party it is very likely that the data coming from that API does not entirely fit the client needs. Consequently, the data has to be transformed on the client side. This data transformation process leads to a heavy client, especially when scaling up the number of APIs to consume. The client then compromises its agility and changeability.
Model-driven approaches support the management of complexity by enhancing the level of abstraction [2]. Also in the field of REST, modelling techniques are used to enhance manageability of APIs and facilitate their development (e.g. [3–5]). In MDSD the development process is driven by formal models which will then be automatically

transformed to code. Thus, the developer can focus on the model construction and does not get carried away with the technical details. Several of the approaches which are modelling RESTful services concentrate on the server-side and automate the development of the APIs.
This work, however, focuses on the client-side by trying to reduce complexity in the client when requesting data from different RESTful services. The aim is to model RESTful API consumption and being able to automate the implementation of this consumption.
The idea is to shift away the responsibility of API consumption from the client and to establish a web service in between the client and the RESTful services which deals with the transformation process. The client then just has to communicate with one API, namely the introduced service. This service provides the data exactly as needed by the client so that no further transformation is necessary and the developer obtains a lightweight client.
The present thesis follows a model-driven approach to automate the implementation of the consumption of RESTful services. Therefore, a model will be introduced which describes the required artefacts to consume RESTful services in single page applications. Subsequently, a transformation process will be presented where an application developer has to specify the transformation rules. The result of this process is a generated service that handles the RESTful service consumption for the client. This helps the developer to focus on what data is needed and where to get it from instead of getting distracted by implementation details. It shifts the service design process to the client, instead of the data owner.
To approach the problem of complex clients whilst consuming multiple RESTful APIs the following contributions will be made:

- Provision of an architectural approach to deal with data that comes from several different RESTful APIs. This allows service composition to retrieve the data as needed. (section 3.3)
- Establishment of a model that describes the client-side consumption of RESTful services. (section 3.4)
- Presentation of a platform which allows constructing such a model and transforms it to code, implementing a web service that can be used to fulfil the suggested architecture. (section 3.5)

1.2 Research Questions

The overall idea is to provide a model-driven approach for RESTful service consumption. The context of this idea is the consumption of multiple APIs in single page applications. On this basis the following research questions were defined to guide the research.

RQ1 What is the state-of-the-art in model-based RESTful API integration?

RQ2 How does a model-based approach for RESTful API integration in single page applications look like?

RQ3 What are the benefits and limitations of a model-driven approach for RESTful API integration?

Research question one targets the literature research, question two the design of an approach and question three the evaluation of the approach.

1.3 Research Methodology

The research methodology that was followed while working on this thesis is the design science approach described by Hevner et al. [6]. Figure 1.1 visualises the different elements of the research methodology. Hevner et al. describe design science as three areas. Firstly, the environment representing the application domain and where the initiation of the problem definition comes from. Secondly, the knowledge base that influences and supports the research with methods, theories, model etc. of previous research. Thirdly, the design science research itself which consists of building artefacts and evaluating those. Furthermore, Hevner et al. explain that the three areas influence each other by three different cycles: the relevance-, the rigor- and the design cycle. Those three cycles will be described in the following by mapping them to the research of this thesis.

Relevance Cycle As stated by Hevner [7], design science research starts with"*... identifying and representing opportunities and problems in an actual application environment.*". The relevance cycle puts the research in the context of the application domain but also provides the criteria for evaluating the designed approach. The application domain in this thesis is web development with the focus on the consumption of RESTful APIs in single page applications. The problem that is being addressed in this environment is the complexity that the clients lead to when scaling up the number of APIs to consume. Therefore, the need is to reduce this complexity and make the consumption of multiple APIs manageable.

Rigor Cycle The rigor cycle contains the scientific part of the process. It connects the actual research with the knowledge base. The knowledge base contains methods, theories, models etc. from previous research projects. With the help of the rigor cycle the artefact development is being supported by the knowledge base

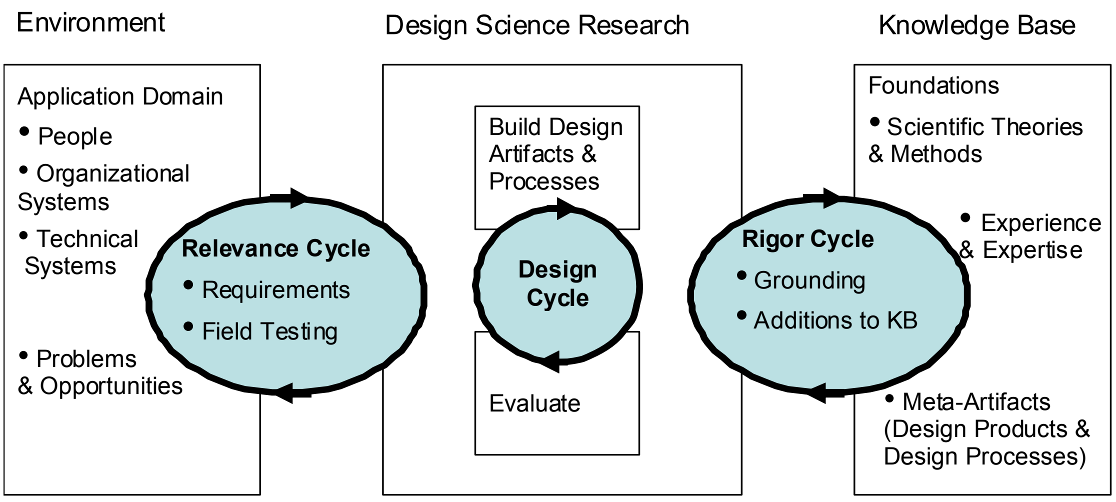


Figure 1.1: Overview of the design science approach [7]

and shows where further research is necessary to broaden the knowledge base. In this work, the knowledge base contains the field of model-driven software development which supports building a solution. Furthermore, it contains related works of modelling RESTful APIs. After the problem was defined at the beginning of the research, the next step was to find out what other people already did in this field. How did other model RESTful APIS? What elements are needed to consume such APIs? Finding out about the state-of-the-art in this research field, forms a valuable basis for building the artefact.

Design Cycle The design cycle is the heart of a design science research project [7]. It consists of iterations between building an artefact, evaluating it, then refining the artefact with the feedback and so on. The developed artefact proposes a solution to the defined problem which is evaluated by getting input from the environment (application domain). In the case of this thesis the design cycle process was as follows: Firstly, a reference architecture was established as well as a meta-model describing RESTful API consumption. On this basis a model-driven approach was designed. After getting feedback from a research assistant, the model-driven approach was refined and implemented as a web application. To evaluate the developed tool, it was used to create an example application (presented in the evaluation chapter). The idea for the application came from a second research assistant, who was targeting this problem at that moment. This gave input for refining the tool and implementing a second example application with the help of the tool.

1.4 Thesis Outline

The present thesis is divided into analysis, design, implementation and evaluation. Chapter two contains important concepts and technologies upon which this work builds. Firstly, the service oriented architecture will be described and the term service will be defined. Following this, REST and ROA will be introduced as well as the consumption of RESTful APIs. The subsequent section deals with the development and description of RESTful APIs. Finally, the chapter analyses single page applications and describes to what problem the usage of such may lead.
Chapter three covers the design of a solution to the previously described problem. Therefore, methodologies and technologies supporting this solution will be presented. Subsequently, an approach will be presented that aims at the reduction of complexity when consuming multiple RESTful APIs. This approach is described by introducing a reference architecture, a meta-model and a code generation process.
The fourth chapter illustrates the realisation of the approach from the previous chapter. It presents the implementation of the code generation tool as well as the generated server. For both software projects the architecture and the dynamic behaviour will be outlined.
Chapter five evaluates the approach by describing two example applications that have been developed with the support of the code generation tool. Furthermore, a comparison between utilising the presented approach and using a conventional framework is made. Subsequently, the evaluation will be analysed by stating the benefits and limitations on the tool.
The final chapter six concludes this thesis and provides an overview of future work that might be useful to enhance the presented approach.

2 Analysis

Contents

This chapter describes technologies and principles upon which the problem as well as the presented approach builds. These topics form the knowledge base and lead to the problem targeted by this thesis. The chapter starts with an introduction to the service oriented architecture and its definition of a service. Secondly, REST and its architectural approach ROA will be presented which follow a different concept to SOA. It will be discussed how RESTful services can be consumed nowadays. Furthermore, technologies supporting the development of RESTful APIs will be presented. Subsequently, single page applications will be described as well as problems coming up when utilising those as a client.

2.1 Service Oriented Architecture

The Service Oriented Architecture (SOA) is an approach to structure an IT system into services. Christudas, Barai and Caselli [8] describe the service-oriented approach as *"identifying the business functions that your applications are made of"*. This statement reveals that the architecture focuses on the business layer. In SOA, components of the software relate to business functions. A business automation logic is separated into smaller units of logic: the services [9]. Thus, a business process can be divided into several steps which then run separately as services in the architecture. They are decomposed but not isolated since they still interact with each other. What characterises those services?

2.1.1 Services

In SOA, services are units of logic which exist autonomously but still relate and communicate with each other [9]. While the size of a service can vary, they follow certain principles. According to Erl [9] those principles of service-orientation are: loose coupling, service contract, autonomy, abstraction, reusability, compostability, statelessness, and discoverability. It is possible to implement services in any language and they also can be web services, however, not necessarily need to be. All services put together form the architecture and they relate to each other because of their communication. In order to enable this interaction between service, they have a service description so that other services are aware of them [9]. In a SOA composed of several services it is also a challenge to manage all these services and handle their communication. For example, the Enterprise Service Bus (ESB) was introduced to address this challenge.

2.1.2 The Enterprise Service Bus

The ESB acts as the backbone of the application and allows integration and management of services in an application. It combines messaging, web services, data transformations and routings to manage interaction of several applications [10]. The services are decoupled from each other and connected through the ESB. It allows connectivity in an application because it enables communication between services by being attached to the ESB. Hence, they do not need to find and connect to every single service themselves. Furthermore, the ESB facilitates combining applications with different technologies. For example, it allows integrating JavaEE applications with the Java EE Connector Architecture (JCA), web services by supporting the protocol SOAP or also .NET applications. Figure 2.1 visualises an example architecture using an ESB and shows how it integrates a variety of technologically different services.

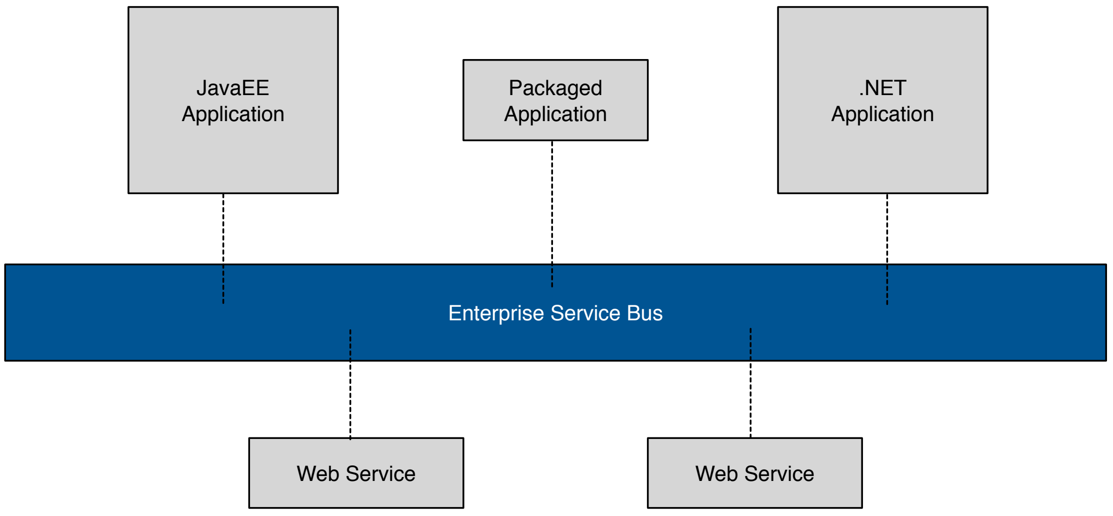


Figure 2.1: Example of an ESB integrating technologically different services (based on [10])

To sum up, in a SOA with several different services, the communication can be managed by the ESB which connects all services with each other and controls the information exchange. It provides routing to the different services as well as data transformations.

2.2 Representational State Transfer

On the internet web clients (e.g. browsers) communicate with servers to load web sites. One way of achieving this information exchange is provided by Representational State Transfer (REST). REST was introduced by Fielding in his dissertation and allows communication between web services with the Hypertext Transfer Protocol (HTTP). Application Programming Interfaces (APIs) are created which define with the help of HTTP request methods if data is being read, written, deleted or modified (GET, POST, DELETE, PUT).
As Richardson and Ruby state in their book [12] REST is more of a set of design criteria than an architecture. That is where the Resource Oriented Architecture (ROA) comes into play as a concrete RESTful architecture.

2.2.1 Resource Oriented Architecture

The ROA follows a different approach than the previously described SOA, however, the two are not incompatible. The idea of ROA is that an application is divided into different resources to represent data. The ROA consists of four concepts and four properties which support applying REST to web services.

Concepts

The main concept of ROA are resources. It is important that those have meaningful names. Furthermore, ROA defines how the resources should be represented and linked together.

Resource The name already reveals that resources are a basic concept of ROA. But what is a resource? Richardson and Ruby [12] define it as *"anything that's important enough to be referenced as a thing itself"*, meaning it can be a document, a picture or a row in a database. Something that needs to be represented in the web application.

Resource Names Each resource is identified by a unique name to make it reachable. This is being achieved by a Uniform Resource Identifier (URI). Such a URI should be well structured and use meaningful names to be comprehensible. The general syntax components of a URI can be seen in Listing 2.1. The scheme and path components are required though the path can be empty [13]. The path identifies the resource for the specific authority.

Listing 2.1: Components of a URI [13]

```
    {scheme}://{authority}/{path}?{query}#{fragment}
```

Representing Resources Computers see the resources just as a series of bytes in order to send them. This is the representation of the resource namely the meta data that represents the resource. Or as Richardson and Ruby [12] specify it: *"... a representation is just some data about the current state of a resource."*.

Linking Resources Together Luckily, one can just navigate through the web without having to remember all the URIs of the resources one wants to see. This is achieved by links between the resources. One resource links to one or more other resources and so on. Google is a good example of linking to other resources. It can be utilised as a starting point to reach the desired destination without knowing the specific URI. Most RESTful services are hypermedia, meaning they not only contain data but link to other resources [12].

Properties

ROA distinguishes itself through four properties: Addressability, statelessness, connectedness, and a uniform interface.

Addressability Each resource is addressable through the URI and all the interesting data of the webpage is represented as a resource so that all necessary data can be addressed.

Statelessness Statelessness denotes that it does not matter when and how the data will be addressed, it will result as the same. No matter if the user navigated through the web application, clicked on the back button or accessed it through the URI directly.

Connectedness The fact that the resources are linked together as explained above.

A Uniform Interface In a uniform interface the HTTP methods (GET, PUT, DELETE, POST) have to be used the exact same way. For example, overloaded POSTs can be problematic since one single HTTP method is being used to serve multiple non-HTTP methods [12]. In such a case the HTTP POST method is being used for some unknown purpose. According to Richardson and Ruby interfaces stop being uniform when the method information cannot be found in the HTTP method.

This thesis focuses on APIs which are truly RESTful and are conform with the concepts and properties of ROA. This fact is important when it comes to modelling RESTful API requests since standardisation is inevitable for a generic model. As Renzel, Schlebusch and Klamma [14] show in their work, a lot of web services which are considered to be RESTful are not truly RESTful. Hence, each time RESTful APIs are mentioned in this thesis, truly RESTful APIs obeying the concepts and properties of ROA are being implied.

2.2.2 HTTP Requests

REST API calls are conducted as HTTP requests. Since consuming RESTful APIs with HTTP requests play a main role in this thesis the main elements of those requests will be presented shortly.

HTTP Methods

Each HTTP request has a method which specifies the operation the sender of the requests wants to be carried out. The GET method is the mechanism for data retrieval [15]. POST creates a new data object and stores it in the database. The PUT method updates data in a database and the DELETE method is being used when it is intended to remove data. There are also other methods such as HEAD, TRACE, OPTIONS, CONNECT but the aforementioned ones are mainly used.

URL

The URL specifies the address of the resource which is targeted by the request. It does not only contain the host and the path but also may contain *URI parameters* which enable passing along values such as an id [16]. Another possibility to pass parameters along is to specify *query parameters*. Those can be used to filter results and should just be used if their content is not secret.

Header

The request header also allows sending parameters along with the request to provide more information. There are several predefined fields which can be used, for example, to provide more information about the context, suggest preferred data formats for the response, sent cookies (*Cookie* field) along or to authorise (*Authorization* field) [15].

Body

The message *body* is used to carry the payload of that request [16]. It is used when more data than just a parameter value needs to be passed along with the request. This is mainly the case when a data object needs to be updated or created (e.g. POST or PUT request).

Response

The HTTP response contains a status code to inform if the request was successful [15]. It also contains a header and a body to pass parameters along. The body contains the data that was requested by the client and the header is in place to pass along parameters not directly related to the data such as cookies or warnings.

2.2.3 Consuming RESTful Services

Web services provide backend functionality to a client. This functionality mostly implements data handling, either persisting data in a database or transferring data from the database. The server provides these service via RESTful APIs which can be consumed by the client using HTTP requests.
In the following, two different approaches of developing web applications will be outlined. Furthermore, it will be explained which one of the two is being supported with the tool developed in the scope of this thesis.

Approach 1 - build your own backend

The architecture of the first approach of developing a web application is show in Figure 2.2. Mainly two computers communicate with each other: a client which provides the user interface, and a web server which carries out computations and sends results to the client [17]. Furthermore, the server is connected to a database to persist and read data. Such an approach is referred to as three-trier computing and a common use case in large-scale systems [17].

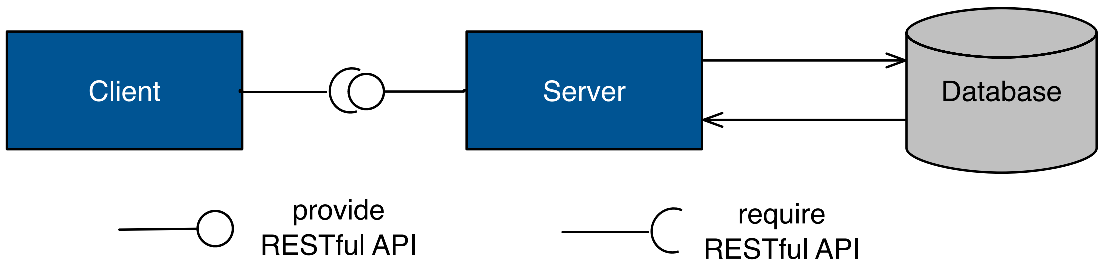


Figure 2.2: Traditional architecture of a web application (related to [1])

The development team is mostly divided into frontend and backend developers. The backend, possibly a Java EE server, is connected to a database (for example, a relational SQL database). It contains most of the functionality and provides data exchange for the web site. In order to make functionality and data from the database accessible for the client, it provides a RESTful API. If the client requests data via this API the backend sends queries to the database to retrieve it. When data needs to be written, the backend creates objects and persists those in the tables of the database.
The frontend mainly consists of templates indicating what the view looks like. When data needs to be displayed in the view the client sends HTTP request to the RESTful API of the backend to receive the required data. Since the purpose of the backend, in such a software project, is to serve the frontend with data transfer and functionality, the API of the backend will be designed as needed in the frontend. In such a case the frontend most likely does not need to conduct lots of data transformation on the response data.

Approach 2 - consume third party services

The second approach of developing web applications is influenced by cloud computing and the fact that more and more software companies publish APIs[1]. Clients are more capable nowadays, leading to the trend of pushing more functionality out to them [17]. In this second approach not everything is being developed from scratch. Third party services are being consumed to retrieve a more meaningful dataset or to profit from functionality already implemented by others. However, these third party APIs have to be consumed as defined by the developers and most likely need to be transformed to fit the needs of the client. The client, possibly a Single Page Application (SPA), contains more functionality to manage all API requests. Current trends in web development such as microservices and serverless architecture influence the architecture of such an application. Before presenting an example of such a web application the important terms are shortly explained.

[1]The ProgrammableWeb.com lists over 18,340 APIs (status as of September 2017)

Single Page Applications A SPA is an application running on the client side which contains just one HTML document. Therefore, the application just has to be loaded once from the server and does not need to be fully refreshed during interaction [18]. The advantage is that navigating between different views of the application works much faster. Those SPA usually also contain a lot more functionality which makes them more powerful and implies that not all functionality is being provided on the server side. (SPAs are explained in more depth in section 2.4)

Microservices The architectural style microservices was first mentioned by [19]. In such an architecture functionality is divided into small web services that run separately from each other. This has the advantage that they can be independently developed and deployed. The client communicates with each microservice separately via a RESTful API. Microservices can be easily added and removed from an software architecture which supports the extensibility of an application.

Serverless Architecture In a serverless architecture clients connect to third party applications which provide ready-to-use functionality (e.g. Backend as a Service (BaaS) or Platform as a Service (PaaS)). For example, those can be services supplying cloud database access (e.g. Firebase[2]) or security and authentication (e.g. Auht0[3]) [1]. Basically, this trend results from cloud computing where servers are not hosted by the web application developers anymore, but outsourced to third parties providing server capabilities (the cloud). Hence, serverless does not exclude servers completely but it implies servers are being managed by third parties and provided as a service.

API Composition API composition is when several services are combined together in one interface. For example, when data represented in a user interface (UI) comes from different APIs. When the UI does not make the API calls directly but has services provide parts of the UI directly and uses these as fragments in the interface, it is called UI fragment composition [20]. Web sites that combine separate data sources and API into one applications are called mashups [21].

These trends lead to SPAs containing a lot more functionality and backend services are not implemented from scratch for each application. For example, microservices that are needed more often (such as login) can be reused across different applications. Additionally, it is quite common nowadays for third party applications, to publish RESTful APIs which allow access to their data and services (e.g. Google, Facebook, JIRA etc.).
An example of an application consuming such APIs is a web application that analyses

[2]`https://firebase.google.com`
[3]`https://auth0.com`

the workload of completed software projects. This example will be used across this thesis and explained in more depth in the evaluation. An organisation wants to have an application that makes predictions on the workload of new software projects based on the data of previous ones. Figure 2.3 shows how an architecture of such an application could look like. Data from each project can be retrieved from the JIRA[4] API since this tool is being used for the project management. The code related to the projects is being stored on GitHub[5], hence, information about the commits and number of lines of code can be extracted from there. These APIs provide lots of different information and not all of it is of interest. Additionally, statistics about the projects have to be calculated based on the project. The data coming from the APIs needs to be combined in one data model. For example, the *Issue* entity in the client holds the description of the issue (JIRA) as well as information about the commit related to this issue (GitHub). Accordingly, several data transformations need to be conducted in the client. To predict workload of future project from the statistics data an estimation service needs to be developed. In order to authenticate the user a cloud based authentication service is being used. The client is a single page application that communicates with all services via RESTful APIs and presents the data in a UI. Hence, the client is much more powerful as it needs to handle all the API consumption and data transformations.

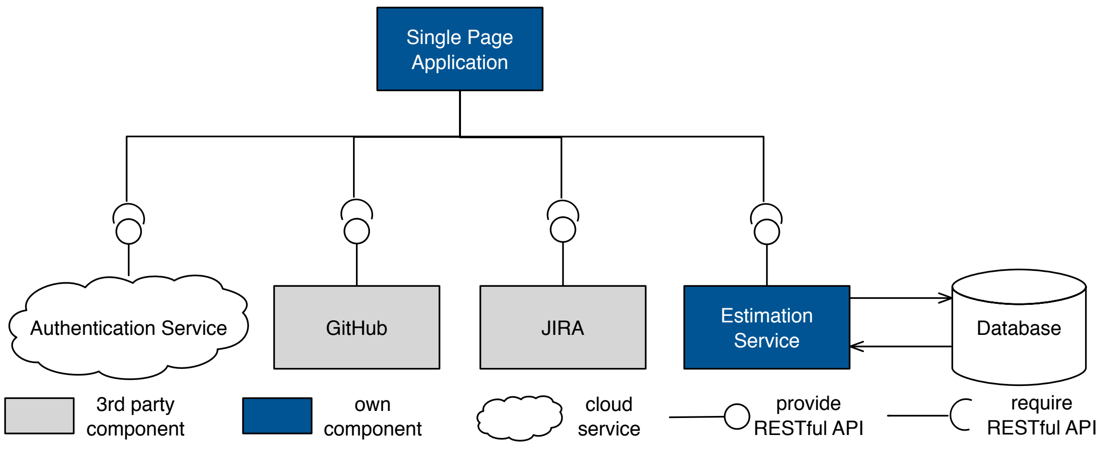


Figure 2.3: Architecture of a web application using modern approaches

[4]http://atlassian.com/software/jira
[5]https://github.com

2.3 Developing RESTful APIs

The fact that it is common nowadays to access third party APIs made it necessary to document the RESTful APIs. If an API is publicly available for every developer it is extremely important that the developers understand the design of the API. Especially in those cases it is required to stick to the concepts and properties of ROA mentioned before. However, it is not enough to just stick to these principles. Developers need to know what resources are available and how to access them. Therefore, most API designers also publish API specifications to explain in depth how to utilise the API. Another way of guaranteeing standardisation and therefore usability for RESTful APIs are API design guidelines. Big software companies publish design guides describing how REST APIs should be designed, also to understand how they implement their software. Furthermore, API management tools facilitate developing and maintaining APIs by supporting each step of the lifecycle.
There is not *the* standard for describing RESTful APIs, however, there are some tools which are widely used. Swagger[6] and RAML[7] are two very popular frameworks to describe RESTful APIs [22]. In the following these two specifications will be presented and compared to make clear what the key concepts of RESTful APIs are.

2.3.1 Describing APIs with Swagger

Swagger is a web development framework to describe, document and test RESTful APIs. The core of Swagger is the OpenAPI specification[8] (aka Swagger Specification) which allows describing APIs [23]. As a language either JSON or YAML can be used. This section outlines how to describe APIs in Swagger. Please note that the specification will not be described in detail but the main objects of the Specification will be presented. For further details refer to the Swagger specification [23]. All information presented here comes from this specification.
Listing 2.2 shows an example of an API description with Swagger in the YAML format. The OpenAPI object is the root of the specification and contains all other objects. In the following it will be described how to specify the general information, the actual resources that are accessible and components describing elements in the specification.

General Information

Firstly, general information about the server and its APIs has to be defined. The `openapi` field is required and specifies the version number of the API specification. Furthermore, an info object and a server object can be defined.

[6]https://swagger.io
[7]https://raml.org
[8]https://www.openapis.org

Info Object The info object (cf. line 2-5) holds metadata about the API. It requires an application `title` and a `version`. Furthermore, it can contain a `description` about the API as well as `license` and `contact` information.

Server Object A server object needs to define a `url` which links to the host. It may contain a `description` of the server and `variables`.

Paths Object

In this object the actual endpoints of the API are being defined. A paths object (cf. line 12) contains several path items and a path is representing the relative path to a resource. If a developer wants to access a resource he/she would just take this relative path and attach it to the url of the server object to target the resource.

Path Item Object A path item object (cf. 13) defines the different elements that are important to conduct a HTTP request as explained in subsection 2.2.2. It contains one or multiple operations objects (cf. line 14) defining what HTTP methods are allowed to access this resource. Therefore, operation objects can be of different types: `get`, `put`, `post`, `delete` etc. A path item object may also contain a parameter object to define `parameters` (e.g. header or query) that are relevant for all different operations.

Operation Object An operation defines how to access a resource with a specific HTTP method. It can also contain parameter objects but if they are listed in the operations object they are just relevant for that specific method (cf. line 19). When describing a POST or PUT request, a request body should be defined. Therefore the request body object specifies how the data for the request body should look like (in the `content` field) and if it is `required`. Defining a responses object (cf. line 26) specifies how the response of this request looks like to inform developers what the different response codes mean.

Parameter Object As already mentioned a parameter object can be defined in the path item object for all operations or in the operations object for a specific one. This object contains the `name` (cf. line 20) and location (`in`, cf. line 21) of the parameter. A parameter can be located in the path, header, query or cookie of the HTTP request.

Components Object

The components object defines objects that can be referenced across the specification (cf. line 35-48). In this way, responses, parameters, request bodies, header, etc. can be defined in more detail.

Schema Object Schema objects can define entities related to the data model of this backend. They can be used as return types (cf. line 34) for requests or as input. A schema object is identified by its name (cf. line 36) and contains several properties (cf. line 38-44) that defines the fields of an entity.

Response Object The response object describes the response of a request to help a developer understand what returning data to expect from an API request. The data `content` can be defined as well as `headers` sent along. It is required to specify a `description` of the response.

Security Scheme Object The operations from the different path can also provide secure requests by utilising a security scheme which is being defined by this object. As a security scheme either HTTP authentication, API key or OAuth2 can be used.

Also several other objects such as parameter, header or example objects can be predefined in the components object and reused in the specification.

Listing 2.2: Example of a Swagger REST API description

```
 1 swagger: '2.0'
 2 info:
 3   title: Person API
 4   description: Simple person API example
 5   version: 1.0.0
 6 host: api.person.com
 7 schemes:
 8   - https
 9 basePath: /api
10 produces:
11   - application/json
12 paths:
13   /people:
14     get:
15       summary: List all people
16       operationId: listPeople
17       tags:
18         - people
19       parameters:
20         - name: limit
21           in: query
22           description: How many items to return at one time (max 100)
23           required: false
24           type: integer
25           format: int32
```

```
    responses:
      "200":
        description: An paged array of people
        headers:
          x-next:
            type: string
            description: A link to the next page of responses
        schema:
          $ref: '#/definitions/People'
definitions:
  Person:
    type: object
    properties:
      first_name:
        type: string
        description: The name of the user.
      age:
        type: integer
        description: The age of the user.
  People:
    type: array
    items:
      $ref: '#/definitions/Person'
```

Of course the here presented fields and objects are not all values that can be defined in the Swagger API description. However, they help understanding what is being described and how to extract relevant information as a developer from such specifications. If a developer wants to access the API, the URL has to be combined with the path. Additionally, it needs to be checked if any header or query parameters have to be passed along and if authentication is required. In order to understand what response to expect the developer can take a look at the `responses` field with the different response codes.

2.3.2 Describing APIs with RAML

RAML stands for RESTful API Modeling Language and allows as the name reveals the specification of RESTful APIs. Similar to Swagger it can also be used to design, build and test APIs. The language to model REST APIs with RAML is YAML. How to describe APIs is defined in the RAML specification [24]. Listing 2.3 shows an example of an API specification using RAML. In the following the main elements of describing RESTful APIs with RAML will be presented. However, not all nodes that can be used for the description will be discussed, for further details refer to the specification [24].

General Information

To define general information about the API, elements such as `title` and `version` need to be defined. Furthermore, the `baseUri` element (cf. Listing 2.3 line 1) can be specified to describe the URL of the API. If parameters appear in the `baseUri` (indicated with `{}`), then those have to be defined in the `baseUriParameters` element. In order to access resources via an HTTP request, the paths defined later in the YAML document need to be added to that base path to option the URL to the endpoint.
Moreover, the general documentation of the API can be linked in the RAML file with the `document` element. It also can be specified what different media types exist and what the default type for response and request bodies is (e.g. XML or JSON).

Resources and Methods

In RAML the endpoints of the API are to defined as a resources property with its allowed methods as child nodes.

Resource Property Resource properties are indicated by starting with a backslash (cf. line 2), followed by the name of the resource. URI parameters can be specified afterwards with the name of the parameter surrounded by braces (cf. line 3).

Methods For each resource the HTTP method needs to be specified to make clear which operations are allowed. For example the code in Listing 2.3 defines in line 4 that a user can be accessed with a GET method. Subsequently other values for a API request can be defined such as a `description`, `queryParameters`, `headers`, or a `body`. For example, line 5 to 8 in Listing 2.3 define four `queryParameters` which reveal that users can be filtered by name, sex and date of birth. A method node can also contain `responses`.

Responses The `responses` node describes the different responses to expect when accessing this resource with the specified method. A response node starts with the response code that indicates if a request was successful or not (1xx Informational, 2xx Successful, 3xx Redirection, 4xx Client Error, 5xx Sever Error [15]). Furthermore, a response can contain `headers` that are sent along with the response and a `body` containing response data.

Listing 2.3: RAML REST API specification of a user resource

```
1  baseUri: http://api.test.com/
2  /user
3    /{username}:
4    get:
    queryParameter:
      name:
      sex:
      dateofbirth:
```

Defining Types

In RAML types can either be defined as JSON or as RAML type. Defining a RAML type has the same syntax as the rest of the RAML and also consists of several elements. It is useful to define types when the aim is to utilise them across the RAML document. They can be used to define the structure of a URI parameter, query parameter, header, or request/response body. Such a definition starts with the `types` node. The next node then specifies the types name, followed by a type and a schema. It can consist of several `properties` which basically define the different fields of a data entity. Those different properties are a combination of the property name and type. A property is required by default but if it is not required it can be stated by `required:false` or by a question mark after the name (e.g. `email?: String`).

When describing a RESTful API with RAML there are also several other properties that can be defined. For example, the authentication and security aspect of an API can be specified in the `securitySchemes` node. Different security types such as basic authentication to pass along a request or OAuth can be defined here.

2.3.3 Key Concepts of API Description - Swagger vs. RAML

The previous explanation revealed specifying RESTful APIs with either Swagger or RAML is quite similar. The differences are mainly naming conventions and slightly different structure in the documents. The key concepts are describing general information about the API, describing the endpoints by listing the different paths (how to access resources and what operations to conduct) and defining types that can be used across the document. These concepts also influenced the process of designing the model for RESTful API consumption which will be described later in this thesis. Even if the model specifies the consumption and not the actual REST API, it makes sense to specify it in a similar manner since the developers are looking at these specifications to find out how to access an API.

2.3.4 API Design Guidelines

As already mentioned before API design guidelines are another way of enhancing standardisation for RESTful APIs. A lot of software companies provide design guidelines which have to be followed by their developers. Some companies also publish their API guidelines so that everyone can access them (e.g. Atlassian [25], Google [26], Microsoft [27]...). These guidelines lack a scientific basis but help software companies and developers to agree on API usability factors [28]. Firstly, the idea to enhance usability for the developers working with those APIs. Secondly, API guidelines should provide a high standardisation for software related to this company. This should make it easier to collaborate between developers which is important when taking into consideration that it is more common nowadays for web applications to access third party code.

Designing Resources

Taking a look at different guidelines reveals that they mainly focus on concepts of REST and resource oriented design. Google [26], for example, explains that the key idea is to represent the data model with the resources and not functionalities. However, each resource has methods attached to it. Furthermore, they state that designing resource should follow a certain process. Firstly, determining what resources to provide and how they relate to each other. Based on this decide how to name the resources. Subsequently, decide on the schema for the resources and use the minimum set of methods for each resource.
Resources can be a single resource or a collection of resources. For example, an *event* resource can consist of several participants (a list of *user* resource).

Naming Conventions

API design guidelines also provide rules on how to name resources. Microsoft [27] for example states in their guideline that the resources names should be easy to read and understand in the URL. In the guideline of Google [26] it says: *"resources are named entities, and resource names are their identifiers"*. Hence, the resource names have to be unique.
However those guidelines do not only focus on the names for the resources but naming in general (also for packages, services, methods etc.). Google [26] specifies that naming should be simple, intuitive and consistent. They provide rules such as using American English, utilising common forms of abbreviations, trying not to use words which might conflict with keywords of programming languages etc.

Further Guidelines

Further instructions that are contained in API guidelines are specifications on how to use methods in REST. Which mainly explains when to use what HTTP method.
Additionally, they also provide a guide on versioning of the APIs. It is being explained

how version numbers work. For example, Google [26] wants their developers to use semantic versioning differentiating between major, minor, and patch version number. Atlassian [25] advises to use the version number of the API in the Uniform Resource Locator (URL). Moreover, they want APIs to support the word `/latest/` in the path, which can be used instead of a version number and specifies the usage of the latest API version.
Furthermore errors are being addressed in API guidelines. Stating how to use error codes, what should be contained in the error messages, as well as how to handle errors.

2.3.5 API Management

API management also aims at supporting the development of APIs. It provides specification languages such as Swagger but goes further and supports each step of an API lifecycle. API management describes the entire process of creating, publishing and maintaining APIs. Tools such as apigee[9] or WSO2[10] provide several functionalities that support API management. Their goal is to facilitate the implementation of APIs while supporting the documentation process and providing analytics. As described by Weir [29] in his book on Oracle's API management solution, there are four fields of API management.

Community Management Community management is an important field for the collaboration between developers. The community of an API consists of internal developers who implemented the API as well as external developers consuming the API. Therefore, it is very useful to describe the API, for example, by using specification languages such as Swagger or RAML. Also other forms of documentation such as providing tutorials explaining how to use the API belong to community management.

API Lifecycle Management The lifecycle management of an API supports it in all phases. Those phases of the API are typically: designing, implementing, publishing, deprecating and retiring. This API lifecycle management is especially important when the APIs are being published, hence can be used by third party developers.

API Operations The third field of API management supports the evolution of APIs through versioning. Furthermore, it includes the presentation of statistics and analytics of the API runtime such as the performance or error rate.

[9] `https://apigee.com/api-management/`
[10] `http://wso2.com`

API Gateway Finally, API management also provides API gateways to allow access to multiple services. Such a gateway can also support the security aspect of the APIs by providing security protocols (e.g. TLS, OAuth).

API gateways provide an interesting architecture which is very helpful when using a lot of different services. Accordingly, they play a relevant role in this thesis and are worth a more detailed description.

API Gateway

A gateway is an architectural pattern that can be seen as a very simple wrapper [30]. API gateways became more used with rising popularity of microservices. They can handle communication with all the microservices for a client or provide various content for different types of interfaces [20]. Figure 2.4 shows an example of an architecture using an API Gateway. The gateway is a layer on top of the server side managing multiple backend calls. It can also vary the content or API calls for different types of frontends. This pattern is for example useful for API compositions to keep API communication in one place and extract it from the client. It is also possible to realise this patter such that each frontend has its own gateways (also called backends for frontedends (BFFs)). This is beneficial if the content for the different interfaces varies a lot and they need to access different services.

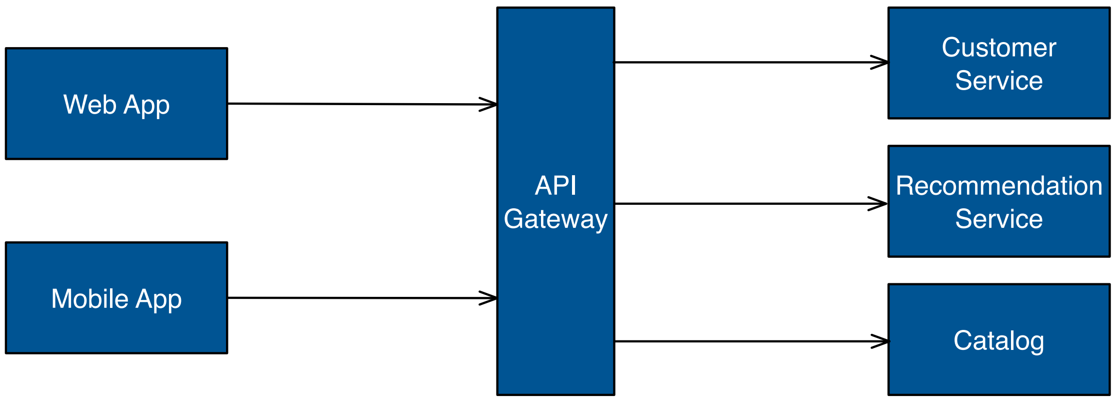


Figure 2.4: Example of an architecture using an API gateway (based on [20])

2.4 Single Page Applications

Single Page Applications are an approach to develop clients in a web application. SPAs rely less on the server and implement lots of functionalities already in the client using JavaScript, HTML, and CSS to realise end-use interaction [18]. Mikowski and Powell

[31] describe them as a fat client that is loaded from the server. The browser loads the SPA once from the server and does not load pages anew for each navigation step. Accordingly, it loads a single page where each view is a sub component and therefore does not need to be request each time form the server. After loading the SPA, the client still communicates with a server to load data which is being displayed in the view, however, it does not load entire pages. Before SPAs were used the client just contained HTML templates and JavaScript for styling. With SPAs the client contains also business logic and HTML rendering. Business and presentation logic is implemented in JavaScript and executed asynchronously in the browser [31].

2.4.1 Building Blocks of Single Page Applications

There are several concepts and technologies that a developer comes across when implementing SPAs. According to Fink and Flatow [18] SPAs have six main building blocks. Those will be presented in the following.

JavaScript Libraries and Frameworks When developing such frontends, several tools and frameworks can be used to support implementing SPAs. Predefined libraries provide templating our routing but also help releasing MV* architectural patterns. For example, the Model View Control (MVC) pattern is such a pattern. The model contains the data, the view represented by HTML templates displaying the model in a UI, the control is the JavaScript code passing the model to the view and handling user interaction. Frameworks such as Angular, React, Vue.js, Knockout.js etc. support pattern such as MVC or MVVM by structuring the application in those different elements. Those frameworks act as the main foundation for SPA and facilitate frontend development [18].

Routing As mentioned before SPAs just contain one page. Hence, such an application cannot just link to different pages to allow navigating through the web page. Therefore, routing is an important feature in SPAs to provide navigation to different views. While staying on the same page routing takes care of things such as navigation history, aligning back button or keeping navigation state available [18].

Template Engines In SPAs templates need to be rendered when a user navigates through the UI. The reason is that when a new section of the page is being loaded not the entire template is requested from the server but just the data. Therefore, the HTML template of the application need to be rendered anew, so that the data can be displayed in the view. Accordingly, templating engines (e.g. Handlebars or Mustache) are being used by those applications. They render context (data) into HTML templates.

HTML5 With the release of HTML5 several APIs supporting the development of SPAs were provided. For example, the history API keeps track of the navigation of the user. Another useful area of HTML5 APIs is offline storage. Some APIs provide, for example, caching or web storage through a dictionary with key-value pairs [18].

Backend API and REST With clients using SPAs also the role of servers changed. They do not provide the entire functionality of the application since a lot of processes can also be handled in the client. However, servers often provide data by being connected to a database. SPA can access this data or persist data by connecting to the APIs of the server. The previously explained RESTful APIs are often used in such an architecture. They enable communication between the client and the server through HTTP.

Ajax Ajax is an important technology for SPAs. It is a combination of JavaScript and XML and uses the XMLHttpRequest object to conduct HTTP requests. Ajax allows code to run asynchronously which is an important feature for frontends. This enables the client to send requests to a server while still staying responsive for the user and handling interaction. Furthermore, it allows rendering of only one section of the UI. The asynchronicity provided by Ajax made applications quicker and therefore it became convenient to put more functionality on the client-side. [18]

2.4.2 Architecture

The general structure of a SPA can be seen in Figure 2.5. The client consists of an application core which contains the different frameworks and libraries which are used by the client. The architecture is structured into different components. The view represents the user interface and specifies what the user can see and how it is structured. It consists of HTML code and uses CSS for styling. The view communicates with controllers (also called view-models depending on if MVC or MVVM pattern is being used). Those controllers are implemented in JavaScript and handle the events triggered by the user interaction. Furthermore, they provide data to the view or pass along data entered by the user. The data is for example stored in a JSON object which represents the model. Data services persist or retrieve those models. To do so they communicate with RESTful APIs of a server or with client-side storage.

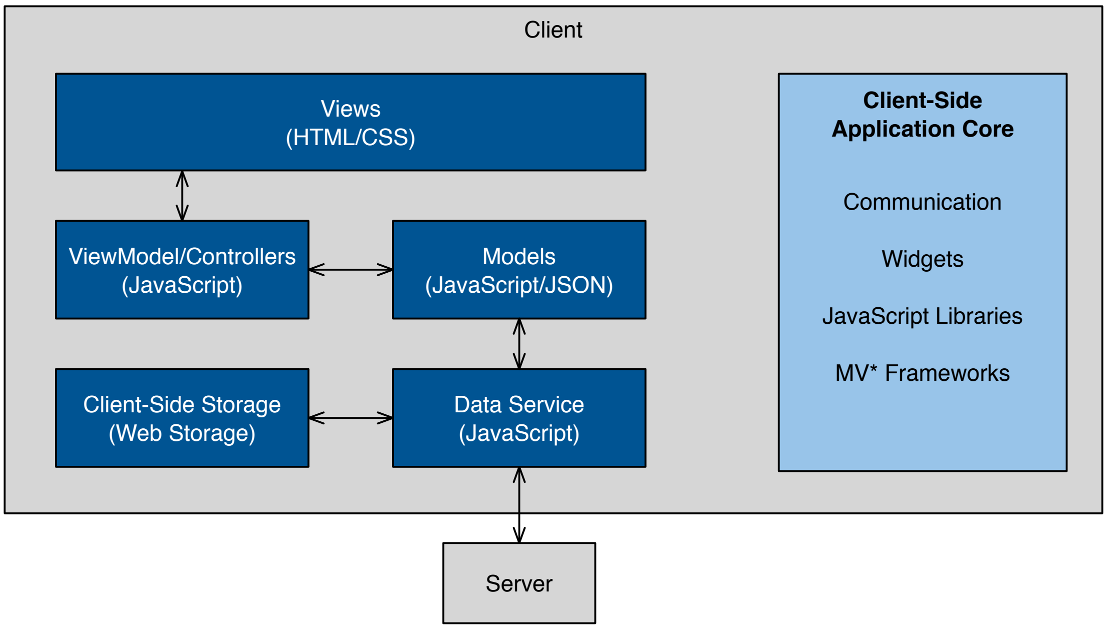


Figure 2.5: Frontend architecture of a single page application (based on [18])

2.4.3 The Problem of a Heavy Client

As previously mentioned, web applications utilising SPAs tend to push more functionality to the client. Obviously, more functionality also implies more complexity on the client side since it contains more code. Another cause for a growth of complexity in the client are data transformations that are related to API consumptions.

It is quite common for clients utilising third party data to transform this data. The reason is that RESTful APIs need to be accessed exactly in the same way as they were designed. This does not mean that the API was designed poorly, it can even happen when strictly following API design guidelines. Data transformations need to be conducted as soon as the data model from the service differs from the desired data model in the frontend. This can happen easily when consuming third party APIs.

An application frontend needs to implement each API request (in the data service) and perform data transformations (e.g. in the controllers) when the API sends the response. In an application which consumes two APIs this is not that much code. However, when scaling up the number of APIs to consume, as it is often the case with microservices, the application will result in having a lot of code in the data services and controllers due to the API requests and subsequent data transformations. This leads to a heavy and complex client which is not very agile or changeable anymore. When a RESTful service is now being changed by its provider, the whole client needs

to be adapted, too. In a monolithic frontend this is not a trivial task. Even though, microservice were used to avoid a monolithic backend it can result in a monolithic frontend and will prevent the flexibility to scale across teams as promised by microservices [32].
Even if the idea is to counteract this monolithic frontend by having multiple clients (e.g. a mobile application and web application) this service consumption and data transformation need to be implemented in all clients.

Approaches Addressing the Problem

The challenge of managing multiple web services consumed by one application is not new and has already been addressed by several other approaches. For example, the ESB which was presented at the beginning of this chapter tries to counteract the problem. It orchestrates several services in a SOA and handles their communication with each other. Furthermore, the previously explained API management tools are also approaching the problem of handling several different APIs. For example, they provide monitoring tools to keep track of all APIs as well as gateways to handle the API consumption.
API management tools do manage complexity but focus a lot more on the actual management of own APIs. This thesis, however, targets reducing complexity when consuming third party APIs. Managing (and evaluating) own APIs requires a powerful tool with several functionalities but in the use cases considered in this work such functionality can unnecessarily enhance complexity. This thesis does not intend to present a better approach than API management tools. The aim is to reduce complexity of the client whilst consuming several different APIs whereas the previously mentioned approaches focused on several other issues. The next chapter presents a further approach to handle client-side RESTful API consumption.

3 Design

Contents

The previous chapter illustrated that the problem of handling and consuming multiple different services is not new. It has already been addressed by architectural patterns such as the SOA or API gateways. This section also presents an approach to handle RESTful service consumption by focussing on the reduction of complexity in the client. Firstly, the research field of Model-Driven Software Development will be introduced as well as approaches and technologies that influence the design of the solution. Subsequently, a reference architecture will be proposed which targets the reduction of complexity on the client-side. Furthermore, a model-driven approach is being presented to facilitate the adoption of the suggested architecture. Therefore, a model is being introduced describing client-side service consumption, followed by a code generation process driven by this model.

3.1 Model-Driven Software Development

MDSD is a discipline where the development process is driven by formal models which will then be automatically transformed to code. Thus, the developer can focus on specifying the instance of a model instead of getting carried away with the technical details of the implementation. This term plays an important role in this thesis since the approach presented is model-driven. Völter et al. [2] introduce the concepts of MDSD in their same-titled book. They state that MDSD is the same as Model-Driven Development (MDD) and Model-Driven Architecture (MDA) fostered by the Object Management Group (OMG) provides the basic terminology for MDSD. Brambilla, Cabot and Wimmer [33] give a good overview of the different fields of modelling by visualising the different subsets of Model-Driven Engineering (MDE) as in Figure 3.1. As explained before, MDD relates to using models to transform them (semi-)automatically to code. The model plays a key role in this process. Völter et al. [2] describe model-driven as a development process where models represent each step: *"... models are no longer only documentation, but parts of the software, ..."*. MDA is a specific version of MDD and provides a standardised modelling language specified by the OMG. MDE is a practice that does not only provide automatisation of implementations but also models other aspects of software engineering (e.g. model-based evolution of the system) [33]. Furthermore, an approach can be considered as model-based when it is concentrated on a model, however the development process is not driven by them. Hence, in Model-Based Engineering (MBE) code is not necessarily being generated from the models.

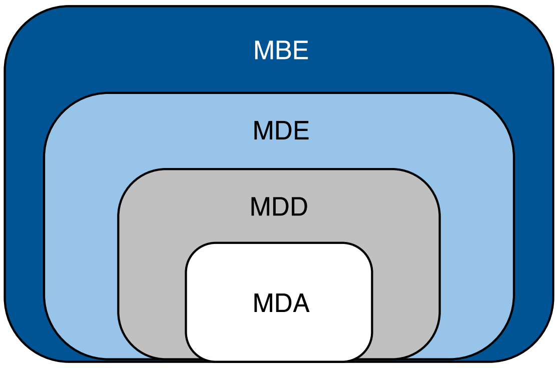


Figure 3.1: Overview of MD acronyms (based on [33])

In this thesis the focus is on MDSD. In section 3.5 a MDSD approach will be presented by implementing a tool which (semi-)automatically generates a server.

3.1.1 Main Concepts of MDSD

A model-driven process starts with constructing a Platform-Independent Model (PIM) which mostly gets transformed to a Platform-Specific Model (PSM), to allow subsequently the generation of actual code. In order to understand the process outline in section 3.5 and why to divide it into these steps, it is important to be familiar with the key concepts of MDSD.

Platform-Independent Model A PIM as described by Lano [34] models the system in terms of domain concepts and constructs independently from an implementation. PIMs pursue the idea: *Concepts are more stable than technologies, and formal models are potentially useful for automated transformation* [2]. Hence, they represent the highest level of abstraction in MDSD and are reusable across different platforms.

Platform-Specific Model PSMs are usually automatically created from PIMs and contain specific concepts of the target platform [2]. They act as an interim step for the code generation to model the implementation of a concrete platform. Such a platform could be a JavaEE server, a NodeJS server, an AngularJS frontend or any other framework.

Model transformation In order to retrieve a PSM the PIM has to be transformed. This process is called a model-to-model transformation. It could also be a transformation from a PIM to a PIM. In the case of actual code being generated from a PIM this process is called model-to-code generation. However, in both cases an artefact is being created from a model and therefore can generally be related to as model transformation.

Figure 3.2 shows the usual process of a model-driven software tool. A PIM is transformed to a PSM which is subsequently transformed to code. However, there can be more than just one PIM as an interim step between the PSM and the code.

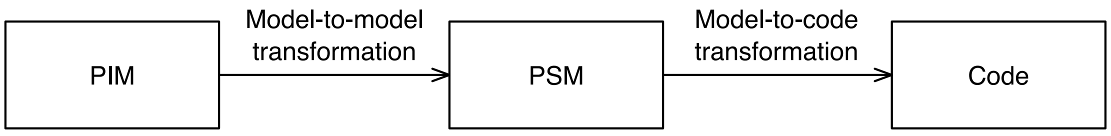


Figure 3.2: Common process in MDSD

The PIM is being modelled with a Domain Specific Language (DSL) and in order to describe how a PIM should look like it can be modelled by a meta-model. The two concepts, DSL and meta-modelling, will be outlined in the following.

Domain Specific Language

A domain describes the field which is being modelled. This can, for example, be a professional domain or a technical field [2]. In the case of this thesis the domain is RESTful APIs and their consumption. A domain specific language allows describing elements of a domain. It specifies the building blocks for modelling by defining the aspects of the models. The developer needs to know and understand the DSL to be able to construct the model correctly. A well known example of a DSL is the Unified Modeling Language (UML) but can also be a newly defined language (however MDA is focussed on modelling with UML-based languages) [2]. In this work models will be described based on UML standards, however for constructing the PIM a web form will be used and not UML.

Meta-Modelling

meta-modelling is being used to describe how a model looks like. This is the basis for presenting a MDSD approach because it allows describing how a PIM has to look like. Völter et al. [2] state that *"...it defines the conctructs of a modeling language and their relationships, as well as constraints and modeling rule,..."*. Furthermore, he outlines that meta-modelling is dealing with several challenges. However, the important two aspects in this work are the following ones. Firstly, a meta-model can be used to describe a DSL to define the abstract syntax this DSL. This aspect will be utilised in section 3.4 when describing the model. Secondly, meta-modelling can be used to describe model transformations to define the transformation rules between two models [2].

Goals of MDSD

The idea of MDSD is to manage complexity by enhancing the level of abstraction. Völter et al. present several other advantages of MDSD. The usage of formal models which are automatically being transformed to code MDSD increases development speed, enhances software quality and allows higher level of reusability. The concept of PIMs ensures a better portability due to platform independence. Additionally, MDSD supports a better maintainability of systems due to redundancy avoidance and provides manageability of technological changes in one place. However, for every concept there are also downsides which are not explicitly being listed here. To compensate for this, section 5.5 focuses on the limitations of MDSD by evaluating the development process of an example web application with the help of MDSD.

3.1.2 Modelling RESTful APIs

After previously explaining the basic concepts of MDSD, this subsection presents the modelling domain of RESTful APIs. Therefore, the field will be outlined by describing different approaches for model RESTful APIs.

API Description Languages

The two API specifications Swagger and RAML that were presented before can be considered as REST API models since they describe RESTful APIs. Similar tools are API Blueprints[1] and Hydra [35] which are also a description language for web APIs. Furthermore, the Wep Application Description Language (WADL) is a language specifically used for the syntactic description of RESTful services which are machine readable [36]. It uses the XML syntax to describe REST principles.

[1]`https://apiblueprint.org`

Those API description languages are also used by model-driven approaches to model APIs. For example Rossi [37] presents a process which starts with constructing a model with UML and transforming it to a RAML description.

The Eclipse Modeling Framework

Furthermore, there are approaches in the field of MDSD which not only model RESTful APIs but also present model-driven approaches to generate RESTful services. A framework which supports the modelling process is the Eclipse Modeling Framework (EMF). For example, Haupt et al. [4] present a meta-model describing REST applications which is being used to automatically generate REST compliant services. They use EMF to model RESTful services.
EMF is an Eclipse[2] framework which supports generating code from models. Models can be constructed with EMF using the UML notation. This UML model can be automatically transformed to XML or to Java code but also from either of those to UML. As stated by [38]: *"Regardless of which one is used to define it, an EMF model is the common high-level representation that 'glues' them all together."*. The model which holds this high-level presentation and therefore unifies the three technologies, is called an Ecore model. The Ecore objects represent a model in memory and is stored in the XMI format [38]. This allows transforming it to either XML, Java or UML.
Such a modelling framework facilitates developing a model-driven approach which is why it is for example used by Ed-Douibi et al. [3]. In their paper they also make use of EMF models and generate them to RESTful APIs. Their process is called EMF-REST and provides ready-to-run web APIs out of data models which are constructed as Ecore models. For this purpose they map EMF to the REST principles so that RESTful APIs can be generated. Figure 3.3 shows an example model of an Ecore model visualised in UML. It describes the structure of a family. This Ecore model is used as the input for EMF-REST and outputs ready-to-run-and-test RESTful APIs [3]. A maven project is being generated from a data model which includes the service implementation classes in Java as well as a simple JavaScript API.

[2]`https://www.eclipse.org`

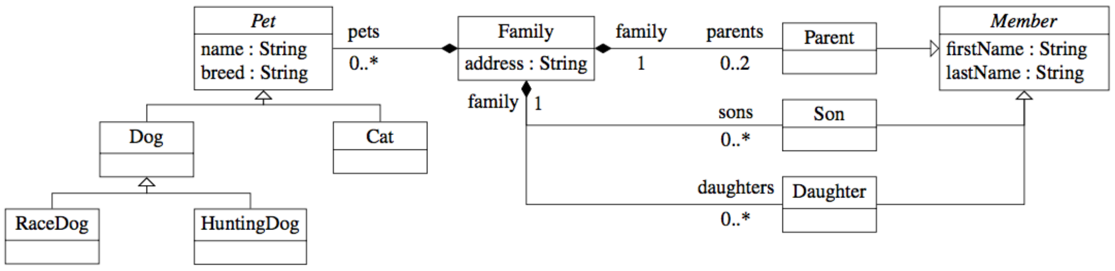


Figure 3.3: Example of an Ecore model [3]

Further Approaches

There are also several other model-driven approaches in the domain of RESTful APIs. For example, Bonifacio et al. [5] present the DSL NeoIDL in their work, to specify RESTful services. NeoIDL can be extended with a program generator to translate NeoIDL specifications to source code [5]. Their provided syntax of NeoIDL is similar to interface descriptions languages of Apache Thrift or CORBA.
Furthermore, Alowisheq, Millard and Tiropanis [39] use UML collaboration diagrams to model Resource Oriented and RESTful Web Services in their paper about Resource Oriented Modelling. Pérez et al. [40] describe ROAs with the *Application Facade Component Model* and Verborgh et al. [41] propose a logic-based approach to describe web API called RESTdesc.
This shows that several other people already worked on model-driven approaches in combination with RESTful APIs. The biggest difference to the approach presented in this thesis is that those works focus on modelling and generating server-side code. This thesis, however, deals with the consumption of RESTful APIs, hence, focuses on the client-side. Nevertheless, looking at models describing RESTful APIs was extremely helpful when modelling the consumption of those APIs.

3.2 Web Technologies

This section introduces web technologies that are being used throughout this thesis. A few frameworks will be introduced to make it easier to comprehend the design and implementation process of the software later in this thesis. Firstly, the JavaScript framework React will be introduced which is being used for the development of the user interface for the code generation process. Secondly, the sever-side tool NodeJS will be described. It is being used as the platform for the generated web service. Finally, the GraphQL framework will be explained to better understand the design decision outlined in the next chapter.

3.2.1 React

The React[3] framework is a library developed by Facebook[4] to implement user interfaces. It is an open source tool which is implemented in JavaScript. React facilitates the development of single page applications as user interfaces. The core of React are components and its compositions which are being realised as JSX files [42].

Components

Chinnathambi [43] introduces React components as: *"... reusable chunks of JavaScript that output (via JSX) HTML elements"*. They contain both, the control and the view. In React templates are not required since components contain everything necessary to be displayed [42]. Listing 3.1 shows an example code snippet of a React component. A React component is implemented as a function and needs to extend `React.Component` (cf. line 3). It also has to implement the `render` function (line 11 to 13). This function contains the view model as HTML like elements, which get rendered in order to be displayed in the view.
React components have properties and a state.

Properties Other components can use sub components and configure them through their `properties`. Those properties cannot be changed from inside the component but from outside [42]. They can be used to pass values to the component which is useful when having nested components.

State Each component has a state which is the part that just can be accessed and changed from inside the component. This inner state can, for example, be used to store input from a user form [42]. If either the state or the properties of a component are being changed, the component gets re-rendered.

In a component events can be triggered. To do so a view element needs to specify the event (e.g. `onClick` or `onChange`). It also has to be indicated what happens if the event occurs, for example, by linking to a function. Such a function will be called when the event gets triggered for the specific view element.

Listing 3.1: Example of a React component

```
1 import React from 'react';
2
3 class ExampleComponent extends React.Component {
4   constructor() {
```

[3] https://facebook.github.io/react/
[4] https://code.facebook.com

```
5        super();
6        this.state = {
7          name: "world",
8        };
9      }
10
11     render() {
12       return <h1>Hello, {this.state.name}!</h1>;
13     }
14   }
```

JSX

React components are implemented as JavaScript Syntax Extension (JSX) files. They combine code to define the UI as well as code to control elements from the UI. In JSX, JavaScript code can be extended with XML-like elements which makes template files such as HTML obsolete [42]. According to Chinnathambi [43], visuals can be defined in JSX files with a syntax similar to HTML but still getting the power and flexibility from JavaScript. The code specifying the view is mainly used in the `render` function of a React component (cf. Listing 3.1 line 12). In order to make the browser read JSX files, it needs to transform the JSX code into JavaScript code [43].

3.2.2 NodeJS

NodeJS[5] is a runtime environment for javascript. It allows writing JavaScript code on every platform where NodeJS can be installed [44]. It can be used as a platform for backends, for example, acting as a server for SPAs such as React or Angular. NodeJS is single threaded which means that it executes one line of code at a time. Satheesh, D'mello and Krol [44] describe the goal that NodeJS tries to solve as follows: *"It tries to do asynchronous processing on a single thread to provide more performance and scalability for applications that are supposed to handle too much web traffic."* Thus, it is asynchronous which means it does not execute the lines of code chronologically from top to bottom but is able to do multiple operations at a time. NodeJS comes along with a package manager.

The Node Package Manager

The Node Package Manager (NPM)[6] is a built-in package manager. It allows installing packages easily with the command `npm install package`. Directories of Node ap-

[5]`https://nodejs.org`
[6]`https://www.npmjs.com`

plications contain the folder `node_modules` were all packages are located which are being used by the application. Packages are organised in the `package.json` file by listing all dependencies that are necessary to run the application. The `package.json` file is important when downloading NodeJS applications from the internet that do not contain the `node_modules` folder. The required packages can then be installed by running `npm install` in this directory.

Express

According to the NPM website, Express[7] is one of the most popular packages installed with NPM. It is a very popular framework for developing NodeJS applications and contains main building blocks and tools to run a server [44]. An Express application contains at least two files. Firstly, the previously mentioned `package.json`. When executing `npm init` in the project directory, such a file will be created automatically. Secondly, an Express applications also contains a `server.js` file which is the entry point for the application [44]. Listing 3.2 shows an example of such a file. It imports the express framework (line 1-2), sets up the server with all its routes (line 4-6), and specifies the port on which the application runs (line 7-9). For this example, the string *Hello World* will be displayed when requesting the URL `http://localhost:8080/`.

Listing 3.2: Example setup of an Express server [44]

```
1  var express = require('express'),
2  app = express();
3
4  app.get('/', function(req, res){
5    res.send('Hello World');
6  });
7  app.listen(8080, function() {
8     console.log('Server up: http://localhost:8080');
9  });
```

The web service (presented in section 3.3) that is being generated with the tool implemented in the scope of this thesis, is realised as a NodeJS server with the help of the Express framework. Its implementation will be outlined in section 4.1.

[7] `https://expressjs.com`

3.2.3 GraphQL

Another technology which will play a major role in this thesis is GraphQL[8]. As stated by Buna [45], GraphQL is a language and a runtime. It is a query language for APIs and a runtime on the server-side to understand GraphQL requests. Later in this thesis the focus is mainly on the runtime since a GraphQL server is being generated in the scope of this thesis. However, the benefits of the GraphQL language lead to the decision to utilise it.
As stated above in REST you have to access resources in exactly the same way as they were designed. GraphQL uses a different approach to transfer data than REST and allows the client to send much more powerful queries to the API than it would be possible with an ordinary REST request. GraphQL is a query language for APIs. The client can exactly specify the structure of a data entity it wants to be returned and does not have to stick to the response structure as it was designed by the backend developer. However, the GraphQL server needs to specify a graph-based schema to define what data can be accessed [45]. The client then sends queries or mutations based on this schema to the server and specifies how the response should be structured. The server responses in the requested format. Thus, the client does not have to transform the data if it wants a different structure as designed by the API. This also leads to less complexity in the client.
In GraphQL there are two types of requests: queries and mutations.

Queries Queries in GraphQL are requests that can be sent to the server to retrieve data of a resource. These queries are read operations and can therefore be compared to GET requests.

Mutations Mutations are API calls that modify data. Every request that is not a read operation is in GraphQL defined as a mutation. In REST, mutations can be compared to POST, PUT and DELETE requests.

As mentioned before, a GraphQL server needs to specify a graph-based schema. It represents the capabilities of the server and describes the supported types, queries and mutations. The schema, allows the server to understand the requests coming from the client and represents the API of the server. Furthermore, a GraphQL server specifies resolvers which retrieve the data (mostly from a database), requested with a specific request. Therefore, each query and mutation also needs to implement a resolve function specifying what the server has to do.

[8]`http://graphql.org`

3.3 Shifting the Responsibility Outside of the Client

This section presents the architectural approach that resulted from the concepts of the previously described methods and technologies. It demonstrates the idea of introducing a GraphQL server between the frontend and the services to reduce complexity in the client. Accordingly, a reference architecture will be proposed for the consumption of RESTful services. Subsequently, it will be justified why a GraphQL server is reasonable by comparing it GraphQL APIs to REST APIs.

3.3.1 Reference Architecture

As mentioned in section 2.4 consuming data from several different APIs leads to complexity in the client. To counteract this complexity the idea is to shift the responsibility of the API consumption outside of the client. Therefore, a so called *query service* will be introduced in the web application which conducts requests to RESTful services. Furthermore, this service also handles the subsequent data transformation. The *query service* provides one single interface for the client and supplies the data exactly as needed by the client. This makes further data transformation in the client obsolete. Figure 3.4 gives an overview of such an architecture.

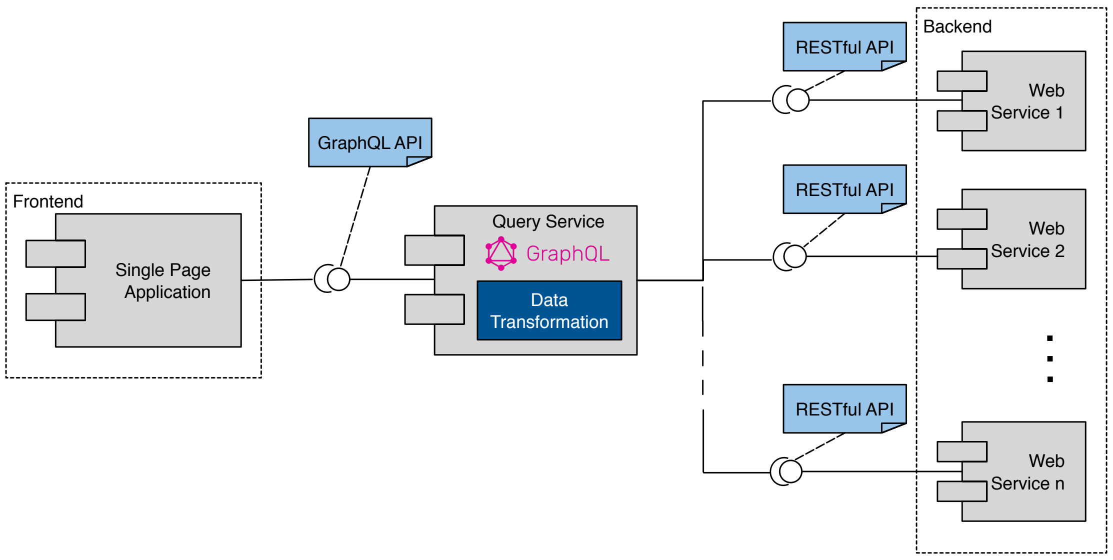


Figure 3.4: Architectural overview of a web application using the *query service*

The *query service* is an instance which runs separately from the client and can be regarded as a web service. It is a GraphQL server to allow the client to send more powerful queries than it would be possible with a RESTful API. For the client it seems

to be the backend of the web application since this is the place to get the data from and the only service to communicate with. The *query service* hides the complexity of accessing several different services from the frontend. It is inspired by the previously described gateway pattern which additionally provides data transformation of the API responses and supplies the data via a GraphQL API to the client.
It provides a GraphQL API to consume RESTful APIs and therefore can be seen as a GraphQL to REST adapter. The advantages of designing the API for the client in such way will be discussed in the following.

3.3.2 Comparing GraphQL with REST

GraphQL is getting more attention when it comes to API development and some even call it the next generation of API design [46]. As both REST and GraphQL have their benefits and limitations this section explains why it makes sense to use GraphQL for the *query service* and not an ordinary REST API.
As Stowe [47] states it: *"GraphQL and REST are designed to solve different challenges"*. Furthermore, he reasons that REST supports having an agile, evolving and version-less API whereas GraphQL is in place to allow querying data models. In REST the language used for the request is different to the one used for the response, whereas in GraphQL those are directly related [45].
When retrieving data from a relational database a developer can exactly specify what data to get from the database, with the help of a SQL query. Requesting data from an API via REST does not provide such powerful querying functionality. Of course it is possible to send query parameters along in the HTTP request but as soon as the queries get more complex REST has its boundaries. For example, when getting the union or intersection of two different fields [48]. When it comes to relationships between entities the client has to take the data as designed by the client. To understand this problem consider the following example:
A user has a name and can be friends with several other users. There are at least two ways to design the API:

- Two endpoints: `/user/{id}` returning a user and its name and `user/{id}/friends` returning the friends of a user (a list of users).
- One endpoint: `/user/{id}` returns the user including all friends.

In the first case, to retrieve the name of a user and his/her friends, two API request need to be conducted. In the second case just one request is needed but for each request all friends of the user are also passed along even if just the name is needed. Which leads to sending a lot of data overhead along with the response which is not being used in all cases.
Listing 3.3 shows how such a request could look like in GraphQL. Here a user's name

is loaded and all his friends. When removing line 4 to 6 just the name will be loaded. It is even possible to go deeper in the user relationship by adding the line `friends{ name }` after the second name field. This allows loading the friends of a user and all their friends within one request.

Listing 3.3: GraphQL query example

```
1 query {
2   user(id: "100") {
3     name
4     friends {
5       name
6     }
7   }
8 }
```

Being able to sent such detailed queries to an API allows the client to specify in more depth what data it wants to be returned. In REST the client does not have any control over the response [45]. This advantage of GraphQL can reduce the data transformations on the client-side, hence, it helps reducing complexity in the client. This is also beneficial because the client knows exactly what to expect as a response since it can precisely specify the structure of the data.
However, there are also scenarios where REST has its advantages, for example, when it comes to the evolution of APIs. It is mentioned as a disadvantage of GraphQL that the API cannot evolve as with REST because of being tightly coupled with the client and changing the API would break the client [47]. This might be an important point when designing APIs of large backends. However, in the case presented in this thesis, the only purpose of the GraphQL API is serving data from other APIs to the client. Therefore, it only needs to be adapted if it is desired by the client. In fact the previously presented architecture supports API evolution really well since the evolution would happen in the RESTful APIs that are being consumed. Hence, the API consumption simply has to be adapted in the *query service* and the client does not have to be touched. It, therefore, will always be fully functional.
To have a clearer idea of this *query service* and how it can be generated as a GraphQL server, the following section describes its meta model, hence, what information it contains and how it is structured. Section 3.5 demonstrates how this *query service* is modelled by the developer and how the subsequent generation process works.

3.4 Modelling RESTful Service Consumption

This section presents a meta-model describing the *query service*. It reveals what information is needed to conduct CRUD (create, read, update, delete) operations on RESTful APIs. To establish this model, client-side applications have been inspected to find out how they handle their data transfer with the backend.

3.4.1 Analysing Established Frameworks

For example, in Angular[9] a usual approach would be to implement an Angular service for each resource. This Angular service then contains functions that conduct CRUD operations to a RESTful API with the help of the `ng-resource` module. A simple example of such an Angular service is shown in Listing 3.4. A service has a function to resolve data (cf. line 7) with a function name and arguments. In order to conduct an API request the service is creating a `$resource` object from the `ng-resource` module. A URL is passed to the resource object (cf. line 3) as well as a URI parameter (cf. line 4). A look at the Angular documentation [49] reveals what further elements could be useful for other cases (e.g. header parameters).

Listing 3.4: Example of an Angular service [49]

```
1  app.service('User', function($resource) {
2    var User = $resource(
3          'http://example.com/api/user/:userId',
4          {userId:'@id'}
5        );
6
7    function get(uId, cb){
8      return User.get({userId:uId}, cb);
9    }
10 }
```

Other frameworks such as NodeJS where analysed regarding API consumption in a similar manner. Additionally, API specification frameworks such as Swagger or RAML which describe RESTful APIs on the server-side were investigated. Taking a look at all these frameworks made it possible to understand what the important elements of API consumptions are. It resulted in a generic model for the consumption of RESTful APIs. This generic model is represented in the model of the *query service* since the main goal of this service is the consumption of RESTful APIs.

[9] `https://angular.io`

3.4.2 The Query Service Model

Figure 3.5 shows the meta-model of the *query service*. A *query service* contains general information about the server such as a name, description, author and on which port it should run for the development process. It contains two other models: a *data model*, and a model describing RESTful service consumption (represented as *resolvers*).

Data Model

The *data model* is specifying the data structure of the frontend. It is used by the *query service* to define the return type of a resolver or an argument type passed to the resolver function. Hence, it is needed to know how to provide the data to the view. The *data model* consists of data *entities* which include several *parameters*, defined by a name and a type.

Resolver Model

A *query service* also consists of several *resolvers*. Those resolvers are functions which represent the API for the client by providing CRUD operations on data to the client. It can be called by the name of the *resolver* function and by potentially passing *arguments* along. Since the *query service* just manages access to several different RESTful services a *resolver* also has one or many *API requests*. This is the part which handles the communication with a RESTful API to provide the data transfer supplied by this resolver. The *API request* contains the *URL* of the API and may include several *query-* and *URI parameters*. Each *API request* also contains an *HTTP Method* (GET, POST, PUT, etc.) describing which CRUD operation to conduct. Where applicable, *header parameters*, a *body* and/or *authentication* need to be passed along with the request.

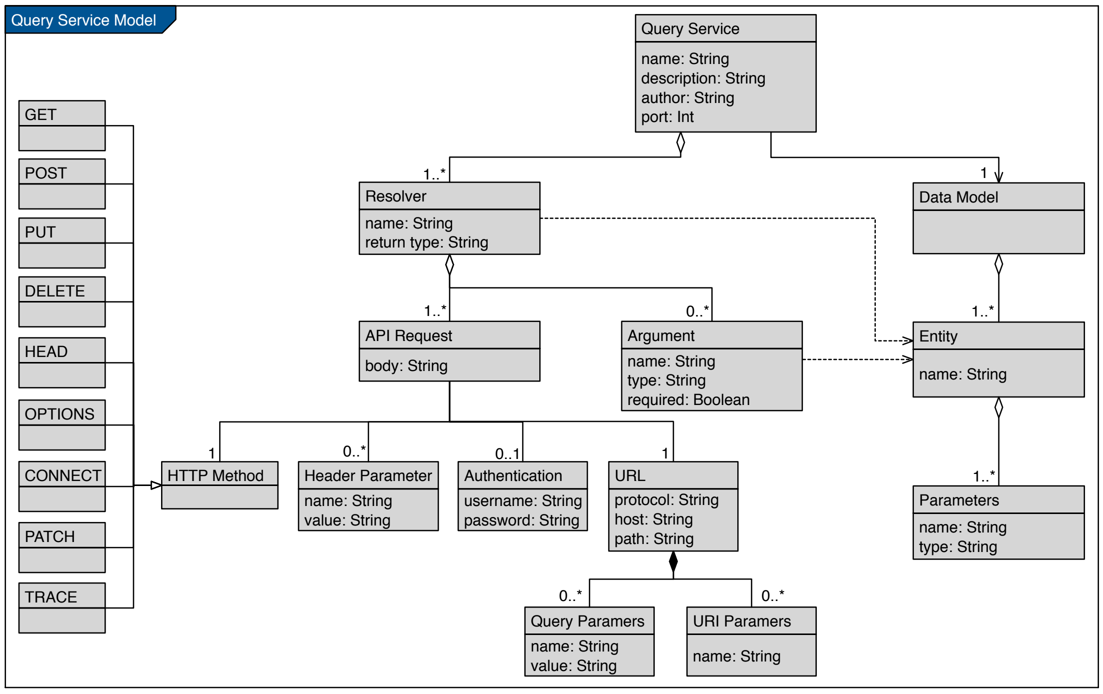


Figure 3.5: Meta-model of the *query service*

3.5 Code Generation Process

To develop a *query service* this thesis presents a semi-automatic process which is driven by the meta-model described in the previous section. Figure 3.6 gives an overview of the process which is accomplished with the help of a web application. It is composed of four steps: (1) construction of the model by the developer using the UI of the application resulting in a PIM, (2) transformation of the PIM to a PSM to support the format of a GraphQL server, (3) code generation based on the PSM, and (4) manual code refinement by the developer.
After justifying the design decisions for the *query service* in the next subsection, each step of the process is being described in greater depth in the following subsections.

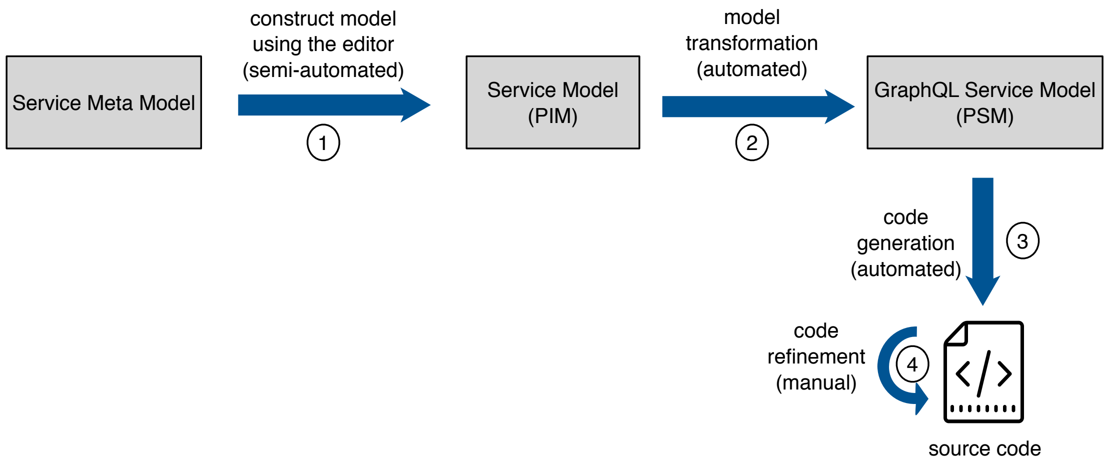

Service Meta Model
construct model using the editor (semi-automated)
1
Service Model (PIM)
model transformation (automated)
2
GraphQL Service Model (PSM)
code generation (automated)
3
code refinement (manual)
4
source code

Figure 3.6: Overview of the model transformation process

3.5.1 Platform Decision

The goal is to provide an implementation of the architecture presented in section 3.3. Therefore, a semi-automatically generated *query service* is being provided. As stated in section 3.3 it is implemented as a GraphQL server. Hence, the client can send queries and mutations to the *query service* in order to retrieve data. The *query service*, however, still uses HTTP requests targeting RESTful APIs. This means it is not required to consume APIs supporting GraphQL. The *query service* can also be seen as an adapter for REST APIs to be able to send GraphQL requests in the client.
As mentioned in section 3.2 the GraphQL client does not have to transform the data if it wants a different structure as designed by the API. This concept supports the idea of complexity reduction in the client and led to the decision using a GraphQL server for this approach. The main elements of a GraphQL server are the *schema* and *resolvers* which define the GraphQL *queries* and *mutations*.

Schema A GraphQL server has to define its schema and resolvers. A schema describes the API of the GraphQL server. It specifies what queries and mutations can be sent to the server by stating the name, arguments and return type of each query/mutation. The schema also contains the data model of the API, describing what resources exist. This is, for example, necessary for the return types of the queries and mutations.

Resolver Resolvers specify for each query and mutation what operations need to be performed. For example, reading data from a database (*query*) or creating a new object and writing it to a database (*mutation*). In the case of the *query service* the resolvers specify what APIs to consume in order to retrieve the requested data.

In the scope of the presented tool, the Apollo[10] framework will be used to implement the *query service*. The Apollo framework defines how to realise and structure a GraphQL server. Accordingly, the concept of both Apollo and GraphQL influence the PSM. But firstly the focus is on the PIM.

3.5.2 Model Construction

The generation process of the *query service* starts with the construction of the PIM. This is a semi-automatic step since the developer has to specify the model according to his/her needs. The model construction is supported by the interface of the web application. The meta-model from section 3.4 is represented as a user form in the interface which then has to be filled out by the developer. For example, the developer has to define the data model and the different APIs to access in order to resolve the data. After the developer finishes constructing an instance of the model it is stored in a JSON object for further process of the application.

3.5.3 Model Transformation

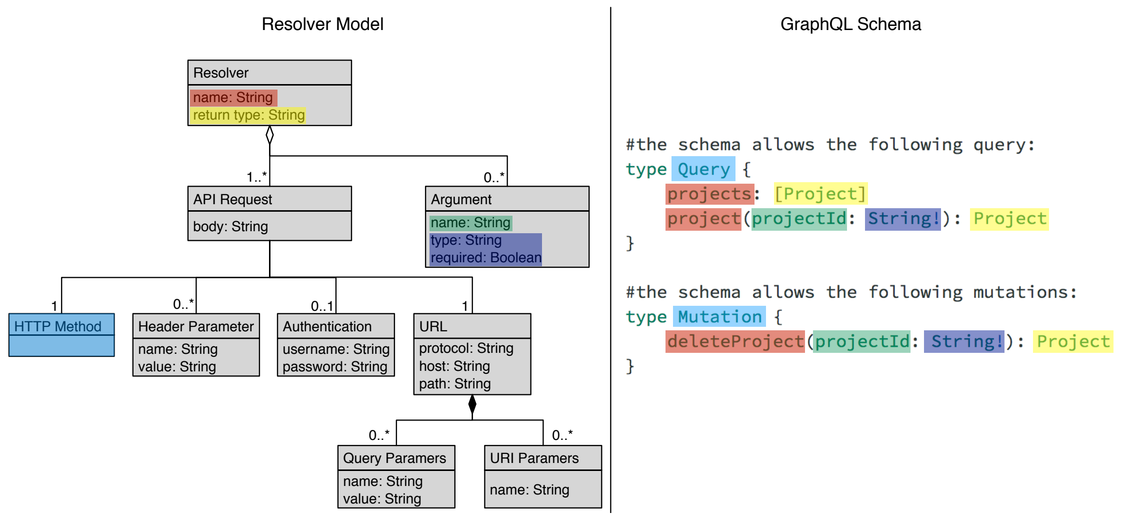
The next step is a model-to-model transformation as the PIM has to be transformed to a PSM. This is just an interim step for the code generation and therefore invisible to the developer. A PSM is based on a concrete platform [2]. In this case the platform is a GraphQL server. As mentioned before, a GraphQL server is specified by defining a *schema* and *resolvers*. The model which was constructed in the previous step, thus, needs to be transformed to be able to generate the platform afterwards.
Accordingly, this *schema* and the *resolvers* have to be constructed in the format as specified by the GraphQL specification [50]. That is why the model transformer consists of a schema-builder and a resolver-builder which both take the PIM as input to extract the information from there.

Schema Builder

A GraphQL schema specifies a data model, queries and mutations. The definition of the data types can be extracted from the data model of the *query service* model (cf. Figure 3.5). Each entity has to be introduced with the `types` key word, followed by the entity name and the list of the parameters.
The queries and mutations definition in the *schema* can be derived from the resolver model of the *query service* model. Figure 3.7 shows what elements to extract from the resolver model to construct the GraphQL schema (related elements highlighted with the same colour). If a resolver is classified as a query or a mutation depends on the HTTP method (blue) of the related API request. If it is a GET method it is a

[10]`https://www.apollodata.com`

query (since no data will be changed) and otherwise it will be treated as a mutation. A query/mutation definition starts with the name (red) and subsequently lists all arguments with the name (green) and return type (purple). If an argument is required it is indicated with an exclamation mark after the type definition. The query/mutation definition concludes with the return type (yellow).



```
#the schema allows the following query:
type Query {
    projects: [Project]
    project(projectId: String!): Project
}

#the schema allows the following mutations:
type Mutation {
    deleteProject(projectId: String!): Project
}
```

Figure 3.7: Related information between the resolver model and the GraphQL schema

Resolver Builder

Information for GraphQL resolvers is extracted from everything related to the resolver model of the *query service* model. Firstly, the resolver is transformed to a function by extracting its name and arguments from the *query service* model. Subsequently, the related API requests need to be constructed. Since NodeJS is being used as the runtime, they are being transformed in the format of `http.request`[11] and `https.request`[12] functions, respectively. Therefore, the HTTP request options are extracted from the model (method, body, authentication, header- and query parameters) and are passed to the request function together with the URL.
After all resolvers from the *query service* model are transformed to resolver functions, they are grouped together and build a new model. This new model is platform specific due to GraphQL and NodeJS.

[11]`https://nodejs.org/api/http.html#http_http_request_options_callback`
[12]`https://nodejs.org/api/https.html#https_https_request_options_callback`

3.5.4 Rendering

The third step is a model-to-code transformation. Alike the previous step the renderer is also an automatic process and therefore, invisible to the developer. Four files (`schema.js`, `resolvers.js`, `package.json`, and `server.js`) are rendered and subsequently exported by the web application. This process is supported by the template system mustache[13]. The mustache templates contain the static content, meaning the code that has to occur in the files no matter how the constructed model looks like. Those templates also indicate where the variable content needs to be placed. The contexts are the variable code parts which are represented by the model from the previous step. Those get rendered into the mustache templates, resulting in the four code files. They contain the following elements from the model of the previous step:

schema.js As the name reveals, `schema.js` contains the GraphQL schema. Hence, it holds the data model and information about what query and mutation functions exist.

resolvers.js The resolver functions which were constructed in the previous step are embodied in this file.

server.js The only information the `server.js` file contains from the model is the port. This file is the core of the server and includes `schema.js` and `resolvers.js`.

package.json The `package.json` file is important to manage locally installed packages for the server. However, it also contains general information about the application from the model such as the name, description and the author.

These four files are sufficient to have a runnable server. The application developer just has to run the commands `npm install` and `npm start` in the directory of the output folder and the GraphQL server will locally run on the specified port.

3.5.5 Code Refinement

In MDSD 100% code generation is possible only for exceptional cases [2]. Thus, such a process almost always includes a step carried out manually by the application developer.
In this process the manual step relates to the resolvers. The code that needs to be added manually specifies what happens after the API request completed. Most likely the developer might want to transform the data coming from the RESTful services. Especially when consuming third party APIs the data model on the server-side differs to the one on the client-side. Accordingly, the received data needs to be mapped to

[13]`https://mustache.github.io`

the data model expected by the client.
It was not possible to model this data transformation subsequent to the API request without restrictions to common use cases. The problem is that the transformation rules need to be specified by the developer. Only he/she knows what fields of the received data belongs to what fields of the target data model. Especially when merging data from several APIs to one entity this seems to be a step that cannot be done automatically.

4 Implementation and Architecture

Contents

The previously described code generation tool automatically creates the *query service* based on a specified model. This chapter presents the architecture and implementation of this tool as well as the generated *query service*. For both software projects, the implementation decisions will be outlined as well as the architecture. It will be elaborated how the design explained in the previous chapter was realised as a software.

4.1 The Query Service

The *query service* is a NodeJS server generated with the process described in section 3.5. It supports the reference architecture presented in section 3.3 and therefore is being used to consume RESTful APIs and provide the received data to a client. Firstly, the frameworks which were used to implement such a service will be explained. Secondly, its architecture will be presented by describing each generated file and its function.

4.1.1 Implementation Decision

As stated in chapter 1 the goal is to reduce complexity in the client when consuming several RESTful APIs. This aim also lead to the design decisions of the *query service* since it should not counteract the issue by adding complexity. Therefore, the following technologies were used to implement the service.

NodeJS For the *query service* the JavaScript based NodeJS was chosen as the platform. It is convenient to configure since it technically just needs two files to run the server (`server.js` and `package.json`). This supports the aim of having a lightweight service. To facilitate the implementation of such a server the framework express was used. Furthermore, the NodeJS libraries `http` and `https` are being used to realise the API requests.

GraphQL Reducing complexity on the client side is also a goal tried to be solved by GraphQL. As previously stated, it allows clients to send more powerful requests than with REST and define more precisely what to expect as a response. Furthermore, the queries prevent the urge to send multiple request for one desired dataset. Using GraphQL can, therefore, reduce the necessity of conducting data transformations on the client side. Hence, the idea was to GraphQL as a runtime for the server, to profit from these benefits.

Apollo To make the server GraphQL based, Apollo framework is being used. It provides a library to facilitate the development of a GraphQL server. Accordingly, the developer needs to specify a GraphQL schema as well as the related resolver functions.

4.1.2 Architecture

Figure 4.1 shows the structure of the folder generated by the tool. It is a lightweight server since it just contains four generated files: `sever.js`, `package.json`, `schema.js` and `resolvers.js`. To run the server `npm install` has to be executed first in the console of the folder. This command downloads all packages that are being used in the code (e.g. `graphql`) and stores them to a folder (`node_modules`). Subsequently, the server can be started with `npm start` and reached on the specified port.

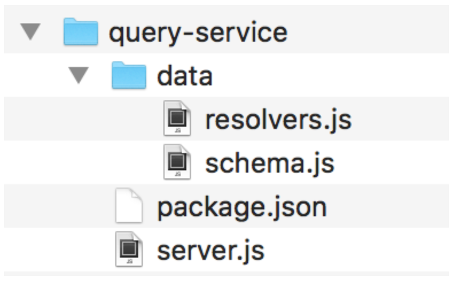


Figure 4.1: Folder structure of a generated query service

Figure 4.2 gives an overview of the architecture of the *query service*. All previously mentioned files are represented in the file except for `package.json`. This file is only important for describing which packages being used by the implementation as explained before. It contains general information about the application as well as package information. However, it does not play a role in the architecture of the running sever.

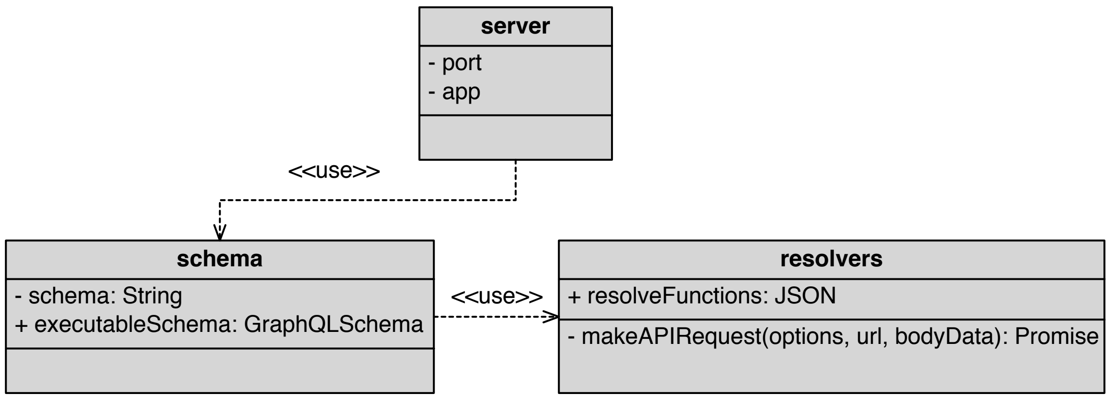


Figure 4.2: Architecture of a *query service*

server.js

This files is the centre of the application since it uses the other two files to create a GraphQL server. It contains the `app` constant denoting the express application and creates it by calling the top level `express()` function [51]. It specifies the port, configures the application as a GraphQL server and defines the routing for HTTP requests.
Technically it would be enough if the server would also contain the schema and resolvers. This would make the other two files obsolete but would make it difficult for the developer, working with the generated code, to read it. Therefore, the decision was to have the GraphQL schema and resolvers in two separate files to enhance readability.

schema.js

The schema file contains a `String` representing the generated GraphQL `schema` which describes the API of the server. The schema defines what entities exist and how to access them. Listing 4.1 shows an example of such a generated schema. It defines the `Project` entity and describes how it can be accessed by the client in order to receive or write data. It is possible to query all projects or a specific one. Furthermore, projects can be deleted as defined by the mutation.
The schema file also imports the resolvers to combine them together with the schema to an executable `GraphQLSchema`. This executable schema then can be used by the `server.js` file to configure the GraphQL server.

Listing 4.1: Example of a GraphQL schema

```
type Project {
  projectId: String
  name: String
  description: String
  duration: Float
  nrCommits: Int
}

#the schema allows the following query:
type Query {
  projects: [Project]
  project(projectId: String!): Project
}

#the schema allows the following mutations:
type Mutation {
  deleteProject(projectId: String!): Project
}
```

resolvers.js

The resolvers file contains the JSON object `resolveFunctions` which holds all the resolver functions generated by the code generation tool. Each query or mutation from the schema is represented as a resolver function in this file to specify where the data comes from. A resolver function implements the access to the APIs, transforms the response data and returns it. Listing 4.2 shows an example of a generated resolver. The example resolve function is called `projects` and returns all JIRA projects (Please note that the example shows a generated resolver that has not been manually refined yet.). Therefore, it accesses the JIRA API to retrieve the desired data. To send an HTTP request to an API (cf. line 13) the resolver file contains the `makeAPIRequest` function which is basically calling the NodeJS function `http.request` and handles its response.

Listing 4.2: Example resolver function `projects`

```
1  projects() {
2    //API request
3    var url = new URL.parse(
4      'https://myexampleproject.atlassian.net/rest/api/latest/project'
5    );
6    var options = {
7      host: url.host,
8      method: 'GET',
9      path: url.path,
    auth: 'username'+':'+'password',
    headers: {}
  };
  var apiReq = makeAPIRequest(options, url.href, '');

  return apiReq.then((resp) => {
    //transform response to JSON object
    var responseJSON = JSON.parse(resp);

    /*
    * TODO: INSERT MANUAL CODE REFINEMENT HERE
    * transform the data from the api response and return it
    */
    return responseJSON;

  }).catch((err) => console.error(err));
}
```

4.1.3 Testing

Once the *query service* is generated and manually refined it should be tested. The server supports, therefore, the in-browser IDE GraphiQL[1] which allows exploring the specified GraphQL schema. It can be reachead by accessing the path `/graphiql` on the specified port. A screenshot of the interface can be found in the appendix in subsection B.2. This testing step is useful to see if the manual code refinement was done correctly but also to test if the specified APIs respond with the data as expected. The GraphiQL testing interface is divided into four components. On the left side of the UI there are two fields to specify the testing input. In the upper field queries or mutations can be entered which is supported by auto completion. The lower field allows the user to specify query variables if necessary. In the middle of the UI the response of the query/mutation will be output. With this output it can be verified if the *query service* works as expected. The right hand side of the GraphiQL UI shows the documentation of the GraphQL schema. All supported queries and mutations can be found there as well as their return types etc.

4.2 The Code Generation Tool

The process of generating the *query service* was already described in section 3.5. This section elaborates how it was implemented as a React application. Firstly, the imple-

[1]`https://github.com/graphql/graphiql`

mentation decisions will be justified. Subsequently, an overview of the architecture will be presented followed by a closer look at each component.

4.2.1 Implementation Decision

The code generation tool is implemented as web application. It contains a user interface by means of which the developer can construct the model. Additionally, the application also implements the model-transformation process to generate the *query service* from the constructed model. For the development of this application the following technologies where used.

React The code generation tool is a React application. The model described in section 3.4 is realised as the UI of the application where each element of the model is represented as a React component. It was decided to use React because no routing is needed for the application and it allows implementing the model component-based.

MDBootstrap For the UI design, the material bootstrap framework was used. It provides ready to use UI components and allows a consistent UI design.

Mustache In the backend Mustache was used as a technology to support the code generation process. The framework conducts rendering of static templates with variable contexts. This functionality was used to transform the PIM to code files. The output of the Mustache rendering process was written to files with the `file-saver` library and compressed with the `jszip` library.

4.2.2 Overview

Architecture

The *query service* creator tool is a React web application which implements the code generation described in the previous chapter. An overview of the architecture of the web application is represented by Figure 4.3.

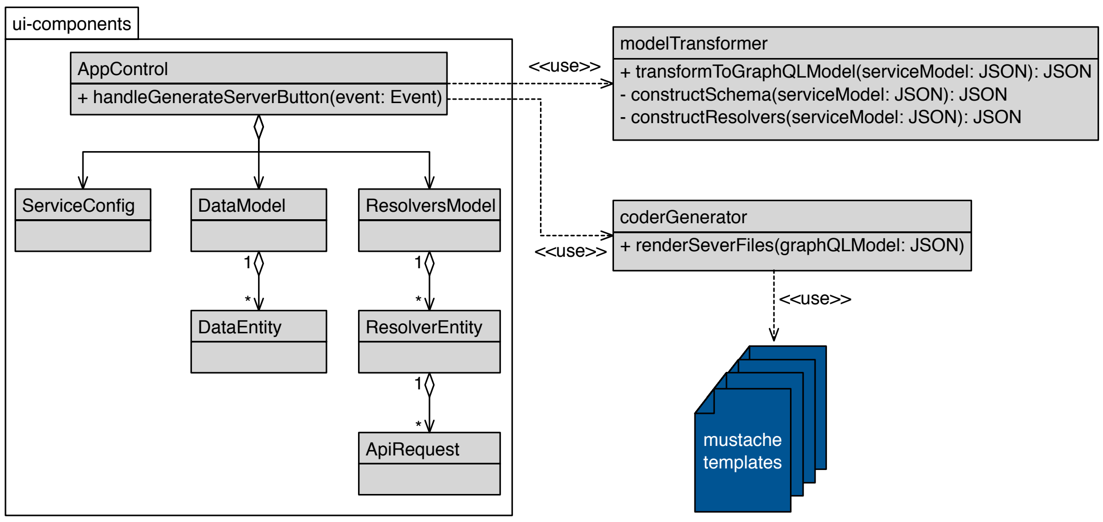

ui-components
AppControl
+ handleGenerateServerButton(event: Event)
ServiceConfig
DataModel
ResolversModel
DataEntity
ResolverEntity
ApiRequest
1
*
<<use>>
modelTransformer
+ transformToGraphQLModel(serviceModel: JSON): JSON
- constructSchema(serviceModel: JSON): JSON
- constructResolvers(serviceModel: JSON): JSON
coderGenerator
+ renderSeverFiles(graphQLModel: JSON)
mustache templates

Figure 4.3: Architecture of the code generation tool

The `AppControl` component is the centre of the application and directs the program flow. The user interface is realised by React components (cf. Figure 4.3 `ui-components`) that represent the different elements of the *query service* model. The components are united by the `AppControl` as one interface. It also handles the event that gets triggered when the developer finished constructing the model. Hence, the `AppControl` communicates with the `ModelTransformer` to transform the PIM to a PSM. Additionally, the `AppControl` component also makes use of the `CodeGenerator` to transform the PIM to a server which uses mustache files to create the code.

Dynamic View

After presenting the static components of the code generation tool, now a dynamic description follows to show how the components interact with each other. For this purpose Figure 4.4 displays the sequence of the code generation.
After the developer finished constructing the model it has to be submitted in order to trigger the code generation process. The `AppControl` component, which handles the submission event, passes the model instance along to the `ModelTransformer`. There it will be transformed into a suitable model for the GraphQL server by constructing the GraphQL schema and resolver functions. Once the model is in the correct format the files can be rendered by the `CodeGenerator`. Therefore, the `AppControl` component passes the JSON object constructed by the `ModelTransformer` which holds the model information (`graphQLServiceModel`) to the `CodeGenerator`. Subsequently, the `CodeGenerator` creates the files (`schema.js`, `resolvers.js`, `server.js`, and

`package.json`), renders the mustache templates with the JSON object, and writes the output to the created files.

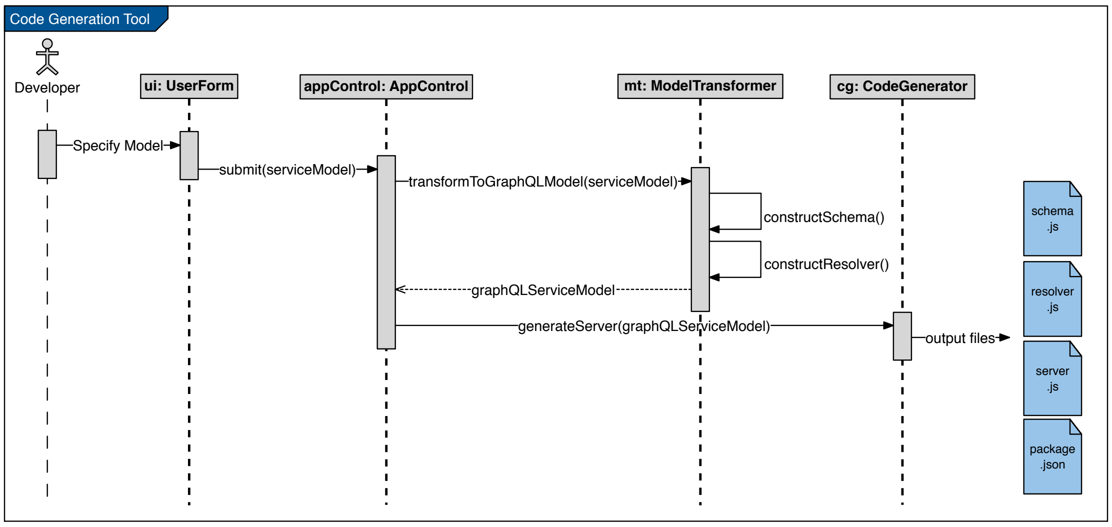


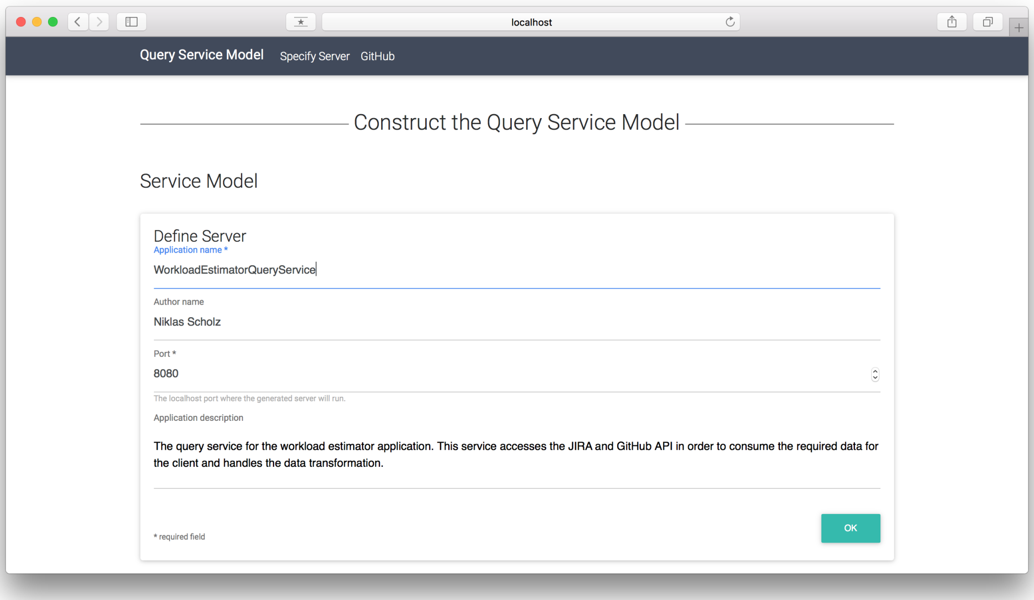
Figure 4.4: The process of the code generation tool

4.2.3 User Interface

The UI consists of React components which are grouped in the folder `ui-components`. These components are `.jsx` files containing templates defining the view as well as the JavaScript functions controlling the different components in the view. Each one of these components consists of a web form which is being filled out by the application developer to construct the related part of the model. The UI consists of three major components: `serviceConfig`, `dataModel`, and `resolversModel`. In the following each component will be described in depth by illustrating the user interface and explaining the different fields.

General Information

The `serviceConfig` component specifies general information of the *query service* and its view is shown in Figure 4.5. The application developer at least has to specify the name of the application (*application name*) and the local *port* where the generated server will run (8080 by default). The fields *author name* and *application description* are optional. Once the fields are filled out, the ok button has to be pressed in order to send the data to the `AppControl` component.



Construct the Query Service Model

Service Model

Define Server

Application name *

WorkloadEstimatorQueryService

Author name

Niklas Scholz

Port *

8080

The localhost port where the generated server will run.

Application description

The query service for the workload estimator application. This service accesses the JIRA and GitHub API in order to consume the required data for the client and handles the data transformation.

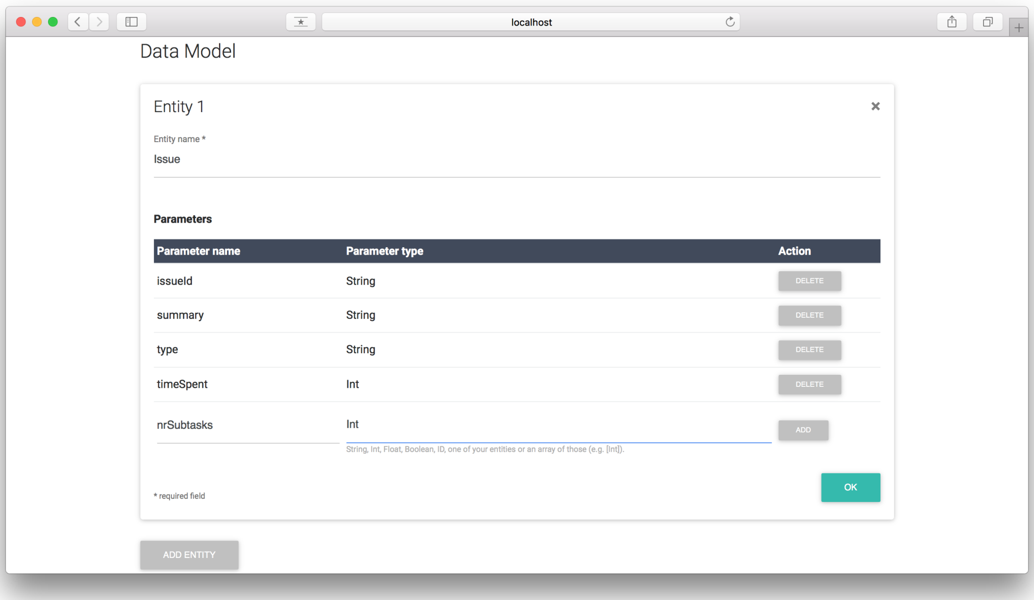
* required field

OK

Figure 4.5: Screenshot of the tool's UI specifying general information

Data Model

The data model of the *query service* is represented by the `dataModel` component. The user can specify the desired amount of entities by adding new ones with the *add entity* button. Therefore, `dataModel` consists of several `dataEntity` components, each one representing one entity of the data model. The user has at least to specify the name of an entity (*entity name*). Furthermore, it is possible to specify any number of parameters related to that entity. For that purpose the *parameter name* and *parameter type* have to be defined. Allowed parameter types are the GraphQL types: String, Int, Float, Boolean, ID or an array of those (e.g. [Int]). Additionally, it is possible to use one of the defined entities as a parameter type and to establish relationships between entities in that way.



Data Model

Entity 1

Entity name *
Issue

Parameters

Parameter name	Parameter type	Action
issueId	String	DELETE
summary	String	DELETE
type	String	DELETE
timeSpent	Int	DELETE
nrSubtasks	Int	ADD

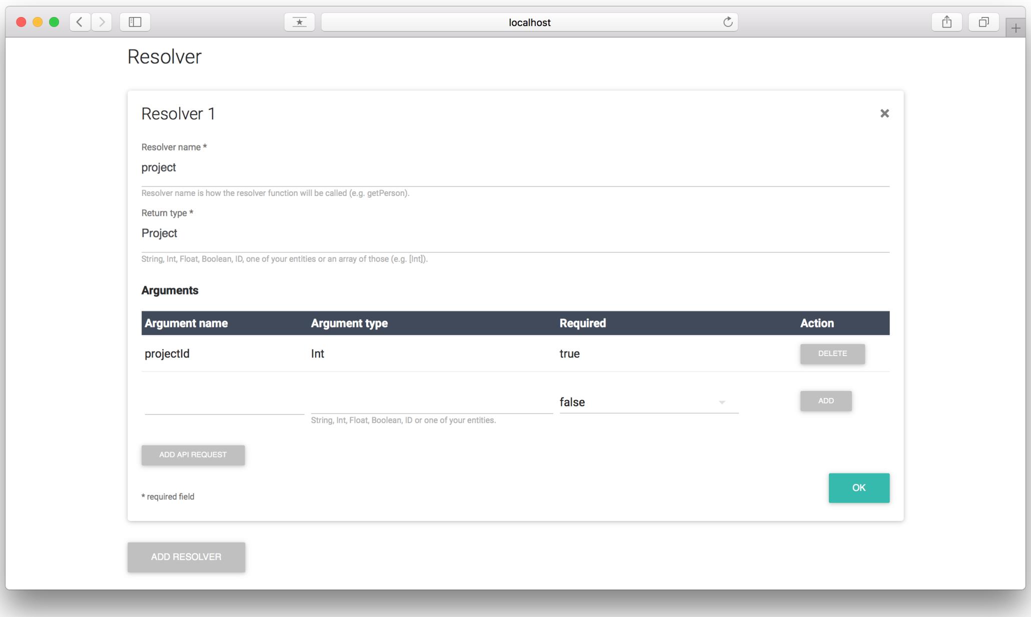
String, Int, Float, Boolean, ID, one of your entities or an array of those (e.g. [Int]).

* required field

Figure 4.6: Specification of a data enity with the code generation tool

Resolvers

The `resolverModel` component allows the application developer to create any number of resolver functions. For that purpose it consists of `resolverEntity` components that can be added by clicking the *add resolver* button. For each resolver a name (*resolver name*) and *return type* needs to be specified. For the return type the same input values are allowed as explained for the data model. Moreover, the user can define arguments to pass along with the resolver. For each argument the name and type needs to be indicated as well as if the argument is required. If an id has to be passed along to retrieve a specific object this would be a required argument, however, if an argument is just in place to filter results it would be optional.



Resolver

Resolver 1

Resolver name *
project
Resolver name is how the resolver function will be called (e.g. getPerson).

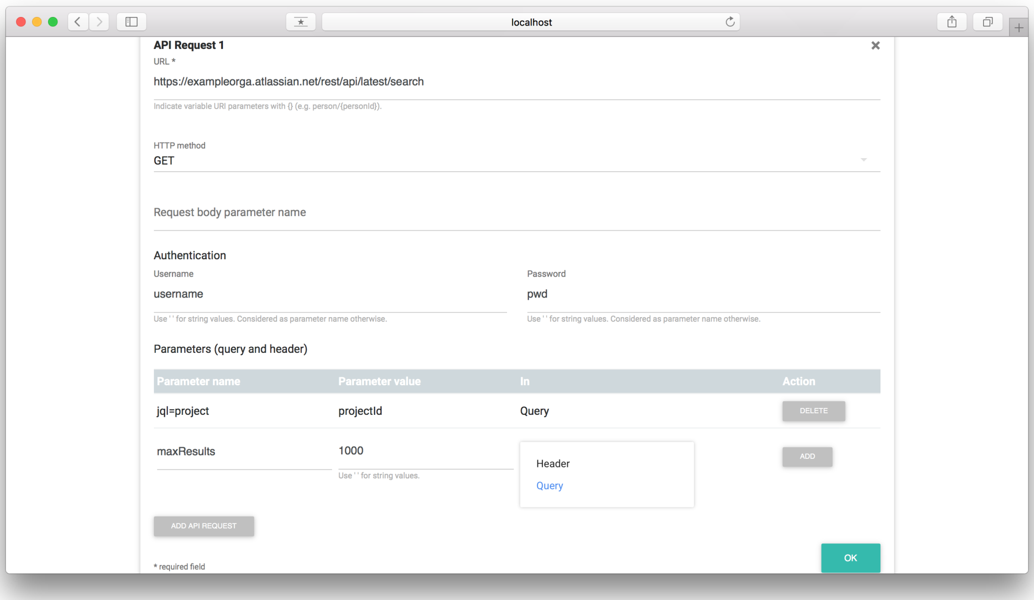
Return type *
Project
String, Int, Float, Boolean, ID, one of your entities or an array of those (e.g. [Int]).

Arguments

Argument name	Argument type	Required	Action
projectId	Int	true	DELETE
	String, Int, Float, Boolean, ID or one of your entities.	false	ADD

Figure 4.7: Specification of a resolver function with the code generation tool

Furthermore, for each resolver function multiple API requests can be defined. Accordingly, a `resolverEntity` component consists of `apiRequest` components. An API request requires the *URL* of the targeted API and the *HTTP method* (GET by default). If the application developer wants to pass URI parameters along with the URL (e.g. the id of the resolver function argument), those need to be indicated in the URL by curly brackets (e.g. /{projectId}). Additionally, a *request body* (for POST and PUT methods) or authentication credentials (*username* and *password*) can be added. If the application developer wants to add query and header parameters to the request, this can be done separately from the URL in the parameter section. For each parameter the name and value has to be specified as well as if it is a header or query parameter. For both the parameter value and authentication credentials there are two use cases. The first case is that the user wants to specify a concrete value, then this should be surrounded by quotes if its a String value. In the second case, the user wants to pass a variable along (for example a value passed to the resolver function as an argument), then the user just enters the name of the variable (without quotes).
Once the construction process is finished the user can click on the *generate server* button to trigger the code generation process.



API Request 1

URL *

https://exampleorga.atlassian.net/rest/api/latest/search

Indicate variable URI parameters with {} (e.g. person/{personId}).

HTTP method

GET

Request body parameter name

Authentication

Username	Password
username	pwd
Use ' ' for string values. Considered as parameter name otherwise.	Use ' ' for string values. Considered as parameter name otherwise.

Parameters (query and header)

Parameter name	Parameter value	In	Action
jql=project	projectId	Query	DELETE
maxResults	1000 Use ' ' for string values.	Header Query	ADD

ADD API REQUEST

OK

* required field

Figure 4.8: Specification of an API request in the resolver with code generation tool

4.2.4 Model Transformation

Once the model was submitted by the application developer, the `appControl` passes along the PIM to the `modelTransformer`. It transforms the model to a PSM and returns it to the `appControl`.
Figure 4.9 shows the dynamic process of the model transformation. In this diagram each function of the `modelTransformer` is being represented as an object. The public function `transformToGraphQLModel` controls the process and constructs four separate models, one for each server file. The transformation to the server and package model is a simple step since relevant information just have to be extracted from the *serviceModel* (step ① and ②).
In step ③ the GraphQL schema gets constructed by calling the `constructSchema` function. It extracts the data model from the service model and creates GraphQL types on from this information. Moreover, it constructs the queries and mutations for the schema by calling the `constructOperator` function. An operator will be added to the schema as a query if the HTTP method of the related API request is a GET method or as a mutation otherwise. After the construction process of the GraphQL schema finished it will be returned.
The most complex step of the model transformation is the construction of the re-

solver functions (step ④). It starts with the call of the `constructResolvers` function. Firstly, it extracts all resolvers from the service model. To construct the resolver functions, it loops through the resolvers, calls the `constructResolverFunction` method, and passes along the resolver. This function then extracts all API requests related to the resolver and creates API request code by calling the `constructAPIRequest` method. To actually construct code for the API request the function extracts the relevant information from the API request and renders them in a mustache template. Once all API requests are returned, the `constructResolverFunction` method also renders the relevant information in a mustache template to create the code for the resolver function. The resulting resolver function will then be returned to `constructResolvers` which collects all resolver functions and returns them as the `graphQLResolverModel`.
Finally, the `transformToGraphQLModel` method puts together all four models and returns them to finalise the model transformation process.

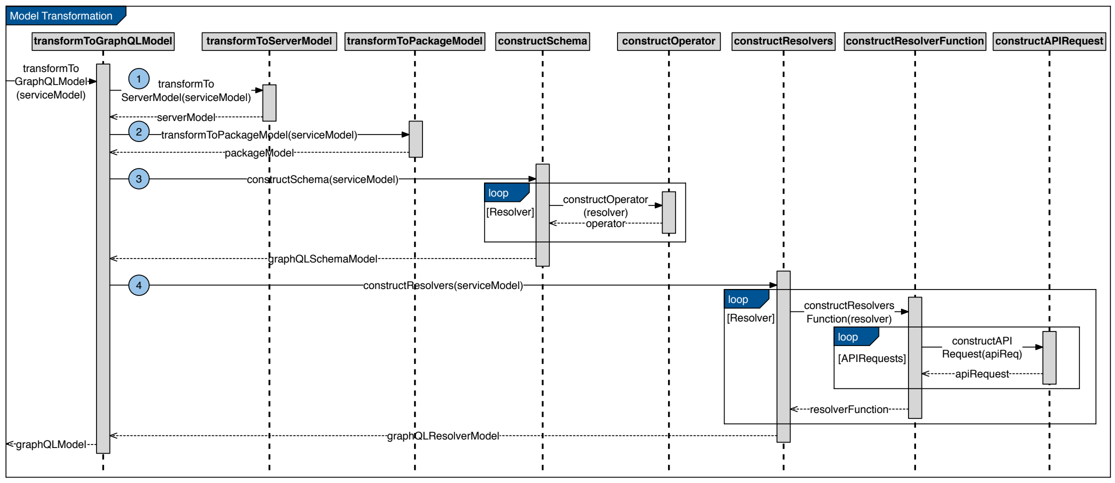


Figure 4.9: Sequence diagram of the model transformation process

4.2.5 Code Generation

The final step of the code generation tool is a model-to-code transformation. For this purpose `generateServer` function of the `CodeGenerator` is being called and the GraphQL model from the previous step is being passed along. The code generator uses the model to render the four mustache templates: `server.mustache`, `package.mustache`, `schema.mustache` and `resolvers.mustache`. The output of the rendering process is being written to the four server files (`server.js`, `package.json`, `schema.js` and `resolvers.js`). Those files are structured as specified at the beginning of this chapter by Figure 4.1. Finally the files are being output as a zip folder.

After this step, the server is fully generated and process of the code generation tool finished.

5 Evaluation

Contents

This chapter evaluates the approach by developing two example web applications with the help of the code generation tool. For each of the two examples, a scenario of an application will be explained, followed by the development process supported by the code generation tool. Subsequently, two frontends will be compared, where one is utilising the *query service* and the other one is handling the API consumption itself. Furthermore, the development process will be analysed and limitations on the code generation tool will be stated.

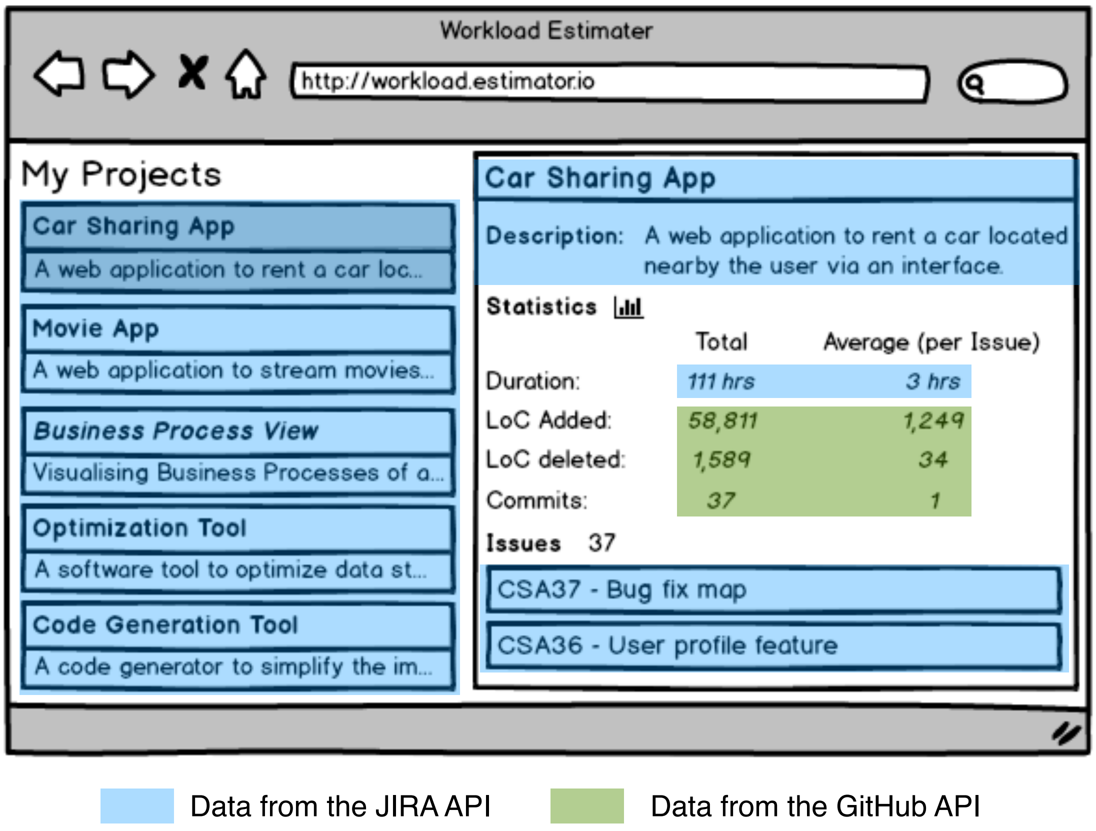
5.1 Example Case 1 - Consuming Multiple APIs

The first example is the web application that was already mentioned in chapter 2. It is an example scenario of the consumption of multiple RESTful APIs in order to retrieve data for the client.

5.1.1 Example Scenario

The idea of the example web application is to provide statistics about completed software projects related to an organisation and make predictions based on this data for future projects.
Figure 5.1 shows a mockup of how a view of such an application could look like. The user interface is structured as a master-detail view and provides a list of all the software projects. When a user clicks on a project, more details and statistics about it will be shown. The project statistics contain values such as the total duration of the project, the average duration per issue, the total number of lines of code added/deleted, and the average number of lines of code added/deleted per commit. Additionally, in the project detail view all issues related to this project are listed. Once clicked on an issue

more details of this issue will be provided such as the number of lines of code which were added/deleted to complete the issue.
Based on the statistics from the projects the app predicts the workload of future projects. General information about the projects and issues can be received from the JIRA API given that the organisation utilises JIRA as a project management tool. Code related data is being provided by the GitHub API assuming the organisation uses GitHub as a project repository.
For the sake of evaluating the code generation tool only the part of the development process which relates to the consumption of the JIRA and GitHub API is being considered. Other development steps such as the user interface implementation are not really relevant here.



Workload Estimater

http://workload.estimator.io

My Projects

Car Sharing App
A web application to rent a car loc...

Movie App
A web application to stream movies...

Business Process View
Visualising Business Processes of a...

Optimization Tool
A software tool to optimize data st...

Code Generation Tool
A code generator to simplify the im...

Car Sharing App

Description: A web application to rent a car located nearby the user via an interface.

Statistics

	Total	Average (per Issue)
Duration:	*111 hrs*	*3 hrs*
LoC Added:	*58,811*	*1,249*
LoC deleted:	*1,589*	*34*
Commits:	*37*	*1*

Issues 37

CSA37 - Bug fix map

CSA36 - User profile feature

🟦 Data from the JIRA API 🟩 Data from the GitHub API

Figure 5.1: Mockup of the example web application and where its data comes from

5.1.2 Development Process

We start the implementation process by constructing an instance of the *query service* model with the help of the tool presented in section 3.5. Figure 5.2 presents an overview of that model instance. To be able to generate the server we need to fill out the user form of the tool.

General Information

First of all, the general information about the web service that is going to be generated have to be entered. Table 5.1 gives an overview of the form fields and their values for specifying the general information. The name of the service, the author's name, the local port on which the service runs for development purposes, and a description of the service have to be entered. The fields application name and port are required.

Table 5.1: General information fields and values specified for the example app

Field Name	Value
Application Name	WorkloadEstimator QueryService
Author	Niklas Scholz
Port	8080
Application Description	The query service for the workload estimator app.

Data Model

Subsequently, the data model for the client side has to be established. To present the data in a web application as stated in the previous section, it is useful to have two entities: *Project* and *Issue* (cf. Figure 5.3).
The *Project* entity contains general information about the project as well as statistics. For example, it holds information about the number of lines of code added/deleted. The project also consists of issues. The *Issue* entity contains information and statistics about a specific issue.
In the user form two entities (*Issue* and *Project*) have to be created. For each entity parameters need to be added by entering the parameter name and type. Valid types are String, Int, Float, Boolean, one of the created entities or an array of those (e.g. `[Int]`).

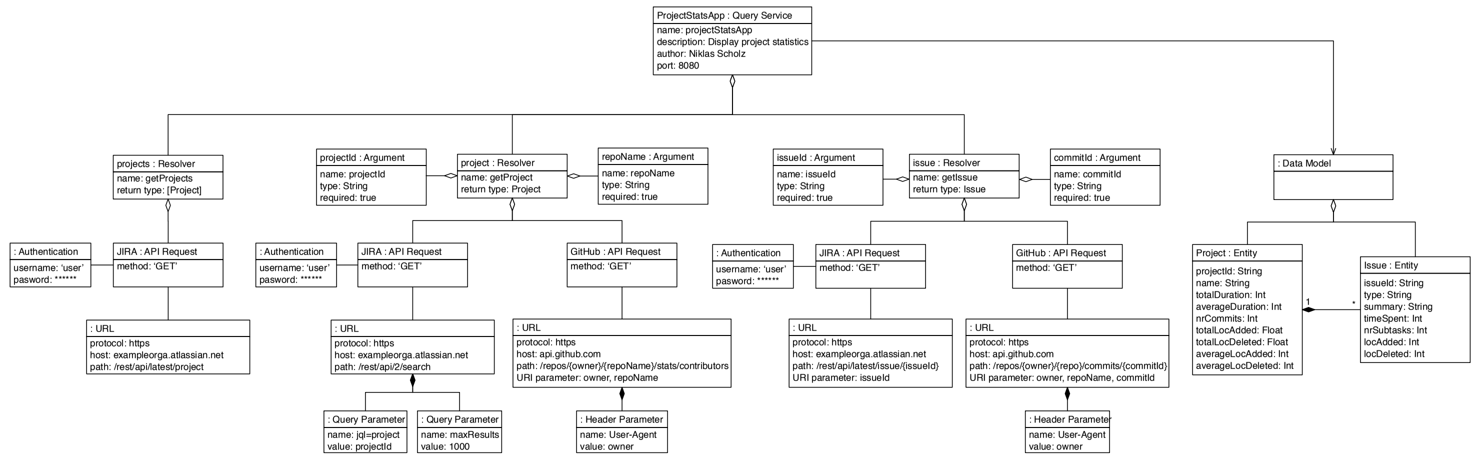


Figure 5.2: Model instance of the *query service* for the example application

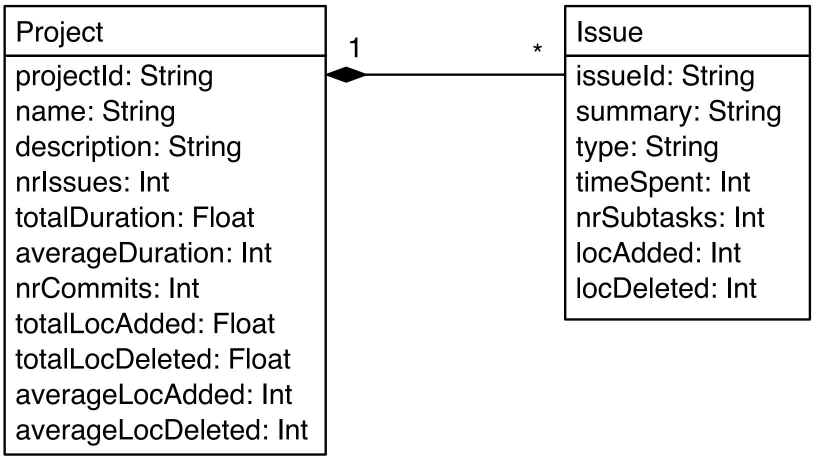


Figure 5.3: Data model of the example web application

Resolver

As a next step, the definition of resolver functions follows to specify from what APIs the data for the entities comes from. Three resolver functions are necessary: `projects` returning a list of all software projects, `project` retrieving data of one specific project including its issues, and `issue` returning detailed information related to that issue. Table 5.2 shows what information related to the resolver functions need to be entered (excluding the API requests). Not all three resolver functions will be explained in detailed but the specification of `projects` will be presented as an example. The values that have to be entered for each API request can be found in Figure 5.2.

Table 5.2: Resolver functions necessary for the *query service*

Resolver Name	**Return Type**	**Arguments** Name: *Type*	**API Requests**
`projects`	[Project]	-	JIRA
`project`	Project	projectId: *String* repoName: *String*	JIRA GitHub
`issue`	Issue	issueId: *String* commitId: *String*	JIRA GitHub

The first values to enter are the resolver name (*projects*) and the return type, an array of projects (*[Project]*). The API request related to the resolver targets the JIRA API. The `projects` resolver is needed for the master view listing all the projects. In this case just the project related values *projectId*, *name* and *description* are needed (cf. Figure 5.1).

In order to retrieve that data, accessing the JIRA API is enough.
The request is a HTTP GET method with the following URL: `https://exampleorga.atlassian.net/rest/api/latest/project`. Username and password need to be entered in order to authenticate to the JIRA server. Query and header parameters do not need to be entered.
The other two resolver functions need to be specified accordingly. After all fields are filled out, the code for the server can be generated.

Manual Code Refinement

To finish the development of the *query service*, the generated code needs to be refined manually. Therefore, code must be added in the resolver functions to specify what happens after each API request completed. Mostly, data which is needed in the client has to be extracted from each API response. In some cases also calculations need to be conducted such as compute the average number of lines of code added per issue.
Listing 5.1 shows the manual code refinement of the `projects` resolver. The response of the JIRA API is being transformed to a JSON object (cf. line 3). Subsequently, for each project of that JSON object the key, name and description is being extracted and added to an array (cf. line 5 to 12).

Listing 5.1: Manual code refinement for the JIRA API response of the `projects` resolver

```
1  apiReq.then((resp) => {
2    //transform response to JSON object
3    var responseJSON = JSON.parse(resp);
4
5    var projects = [];
6    for(var i in responseJSON){
7      var project = {
8        projectId: responseJSON[i].key,
9        name: responseJSON[i].name,
10       description: responseJSON[i].description
11     };
12     projects.push(project);
13   }
14
15   return projects;
16
17 }).catch((err) => console.error(err));
```

Now the server is in place and can be started to handle the communication with the APIs. The next step is the development of the client which only needs to access the established GraphQL server to receive the data. The view can display this data without further transformations since it is being returned exactly as needed.

5.2 Example Case 2 - A Series of API Requests

The second example case deals with the same application as the first one. The only difference here is that the focus is not only on accessing multiple APIs but also on the sequence of those requests.

5.2.1 The Chronological Order of API Requests

The problem of the chronological order of API requests occurred while working on this thesis. The first example dealt with the consumption of two APIs in order to retrieve data. If the either the JIRA or the GitHub API request is conducted first does not matter. However, there are cases where it does matter which API to access first. For example, when the response data of a first API request is required for a second API request. Such a case where it does matter in which order the API requests are sent will be further referred to as *the chronology of API requests*.
Encountering this problem lead to the question how to model such a sequence of API requests and how to handle such cases in the code generation tool. The chronology of API requests cannot be described by a static model such as the *query service* model. It cannot be represented which API request to conduct first, since static diagrams do not represent time [52]. It can describe which APIs to access but not in what order since this is a dynamic aspect. Therefore, to describe the chronology of API requests a dynamic model is needed which describes the query service interacting over time [52]. The UML sequence diagram allows modelling this dynamic aspect.
How the chronological execution of API requests can be realised in the code generation tool will be shown in the following by developing such an example case.

5.2.2 Exampe Scenario

The scenario is the same as in example 1. The only difference is when requesting a specific project or issue the whole data entity should be returned without needing to specify the GitHub repository name or commit id. To still be able to link the JIRA data with the correct GitHub data imagine the following scenario:
The organisation utilising the web application have the guidelines to connect JIRA projects to GitHub repositories. Therefore, each JIRA project contains the URL to the related GitHub repository in the link field of the general information about the project. Furthermore, the organisation has made the restrictions to put the URL of the GitHub commit in the link field of the related issue when it is completed. In this way the JIRA

data is linked to the GitHub data but in order to know what GitHub resource to access a request to the JIRA API is required.

5.2.3 Development Process

The implementation process of this example corresponds to the process of the previous example (same general information, data model etc.). The only steps that need to be carried out differently are the specifications of the `project` and `issue` resolvers. Since they have to be realised in a very similar manner, just the specification of the `project` will be presented as an example.

Resolver

To define the `project` resolver, firstly, the name (*project*) and return type (*Project*) need to be entered. In order to be able to retrieve the correct project, the *projectId* needs to be specified as an argument for the function. Since the JIRA project key is being used as the id, the type is a *String*. Subsequently, the specification of the APIs to consume follow, in order to fetch the data related to the project. Three API requests are required and need to be carried out in a strict order to retrieve all data. Figure 5.4 shows the sequence of the three API requests.

- The first API request is targeting the *issues* resource of the JIRA API. To just retrieve the project related issues the projectId (the function argument) is being passed along as a query parameter. Entering authentication (username and password) is also required to be able to access the data. From the response the following data for the project entity can be extracted: name, description, issues, nrIssues, totalDuration and averageDuration.
- In order to get the missing data from the GitHub API the link to the GitHub repository needs to be retrieved. This can be done by another request to the JIRA API. This time the target is the *project* resource. Therefore, the projectId has to be passed along as a URI parameter (in the URL indicated with `{projectId}`). From the URL linking to the GitHub repository the owner and name of the repository can be extracted. These two values are necessary for the third API request to retrieve statistics about the repository.
- The third API GET request targets the *stats* resource of the GitHub API, passing along the repository owner and name as URI parameters. From the response we can calculate the missing fields (nrCommits, totalLocAdded, totalLocDeleted, averageLocAdded and averageLocDeleted) for the requested project and finally return the project object.

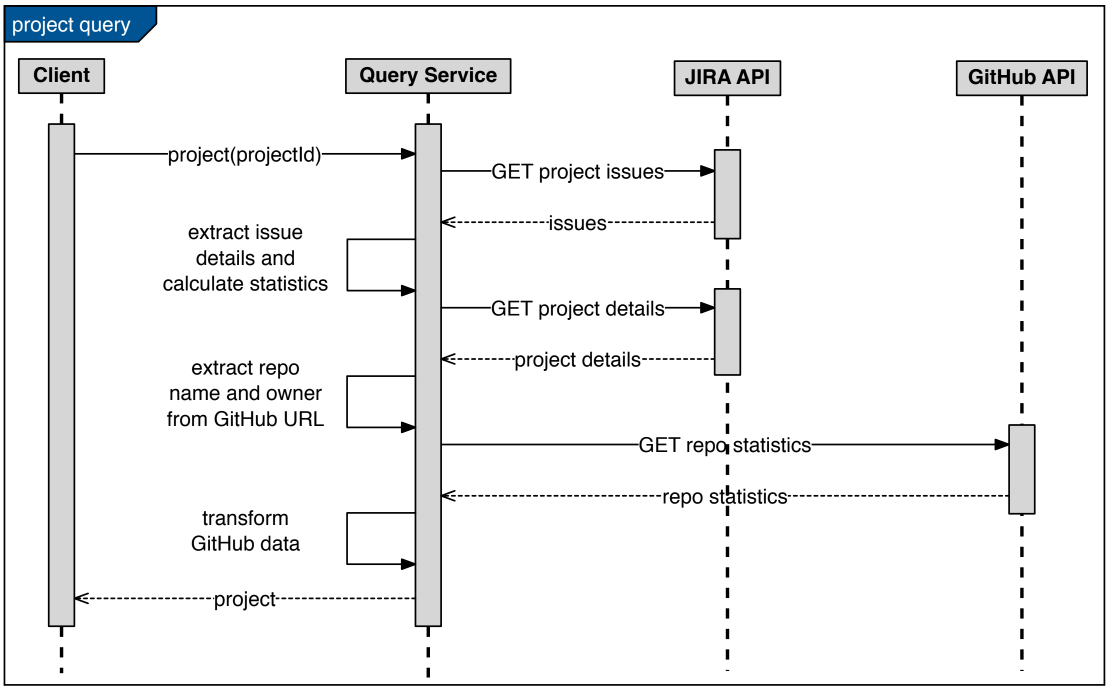


Figure 5.4: Request sequence for the resolver function `project`

Table 5.3 gives an overview of how to fill out all the fields in the UI, related to the API requests of the `project` resolver. It is very important that they are specified in the exact order since the code will be generated so that API request one will be executed before API request two (and two before three etc.). In this way the problem of the chronological order of API requests can be solved by the code generation tool. Listing 5.2 shows an example of how the code is being generated in such a case. It is being realised by chaining promises where the promises are API requests. Only after the first promise resolved, the second promise gets returned and will subsequently be settled. In this scenario this means that after the first promise resolved, data will be extracted from the response and used for the second API request (cf. Listing 5.2 line 1 to 4). The second API request will be returned (line 5) and once it is resolved the data can be extracted from that response (line 8).

Listing 5.2: Example code of two API requests with one being dependend on the other

```
1  apiReq1.then((resp) => {
2    // extract data from response to use it for next API request
3    var respData = extractRelevantData(resp);
4    var apiReq2 = makeAPIrequest(respData);
```

Table 5.3: APIs that need to be consumed for the `project` resolver

API req. number	**URL**	**HTTP Method**	**Body**	**Authentication**		**Parameters**		
				Username	Password	Name	Type	In
1	https:// myexampleproject .atlassian.net /rest/api/latest /search	GET	-	usrname	*****	jql=project maxResults	projectId 1000	Query Query
2	https:// myexampleproject .atlassian.net /rest/api/latest /project/{projectId}	GET	-	usrname	*****	-	-	-
3	https:// api.github.com /repos/{owner} /{repoName}/stats /contributors	GET	-	-	-	User-Agent	owner	Header

```
  return apiReq2;
}).then((resp) => {
  //do something
  var data = extractRelevantData(resp);
  return data;
}).catch((err) => console.error(err));
```

However, if multiple API requests need to be conducted which do not depend on each other this approach is not recommendable. API request two would always wait for request one to settle before being carried out and hence waits for API request one. A better solution for such a case would be to use `Promise.all([apiReq1, apiReq2])` to resolve the promises. In this way the requests can be carried out in parallel.

Manual Code Refinement

After the model instance was constructed and the server generated, the code has to be refined manually as a final step. Also in this example, code needs to be added in the resolver function to specify what has to happen after each API request completed. For example, in the `project` resolver function for all three API requests, data transformations need to be added manually:

- After the first API request completed all issues related to the project need to be extracted from the response and added together with the project name to the project JSON object. Furthermore, on the basis of all issues the average and total duration of the issues needs to be calculated.
- The required part of the API response is the URL to the GitHub repository. From the URL the name and owner of the repository can be extracted. Those two values then need to be added to the third API request.
- The response of the third API request is important to calculate the missing fields regarding the commits and lines of codes. Subsequently, those values need to be added to the project object which then can be returned.

After manually refining the code, the server is fully functional. As a next step the user interface can be developed which receives the data from that server.

5.3 Comparing the Approach to Conventional Frontend Development

This section compares two frontends with each other. One of the frontends belongs to an application using the *query service* for the RESTful API consumption. The other is a

frontend developed in the conventional way, handling the RESTful API consumption itself by using Angular services. The goal is not to evaluate the development process as in the previous sections but to compare the architectures of the frontends to see if using the reference architecture really reduces complexity in the client. Therefore, not the entire architecture of the application will be compared but simply the architecture of the client.
The example application that has been implemented here with two different approaches, is the same scenario as for the previous sections. However, the scenario has been slightly simplified. The application consists of two different views: a master view listing all projects, and a detail view showing a specific project. It was decided not to implement the detail view of an issue to make the example smaller and hence easier to compare.
In the presented architecture not every single file is represented but just the components which are necessary to understand differences. For example, files such as `index.html`, the `app.module` or the `app.component` will not be shown since they do not add value to the comparison and are present in both frontends anyway.

5.3.1 Implementing RESTful API Consumption as Angular Services

The conventional approach to develop such a web application is, for example, implementing it as a single page application with the Angular framework. Angular is based on the MVC pattern, hence, such a frontend can be divided into three components [53]. As outlined in section 2.4 in SPAs the three components are: views presenting data, controllers handling user interaction from the UI as well as transferring data between service and view, and data services requesting data from an API.
Figure 5.5 displays the frontend architecture of the example scenario implemented as an Angular client. Obviously, the architecture could also be designed in a different way than presented here. This is just an example and one way to do it. As the figure reveals there are two views: the `projects-list.view` lists all projects, and the `project-detail.view` shows more information about a project when clicking on a specific project from the list. Both views present data in their view through Angular's data binding by being connected to a controller. For this purpose, the `projects-list.controller` contains a `projects` variable and the `project-detail.controller` contains a `project` variable. When the `projects-list.controller` is being instantiated, the constructor calls the `getProjects` function to retrieve a list of all projects from the data service. Therefore, the `jira-data.service` contains a `getProjects` function which sends an HTTP request to the JIRA API. It is enough to call the JIRA service, since for the project list just the ids and names of the projects are needed. `jira-data.service` returns a promise containing the response data to the controller. The response data is not exactly structured as needed for the view, hence it needs to be transformed by the

`transformProjectData` function.
When a user clicks on a concrete project from the list the `project-detail.view` will be displayed. Therefore, the `project-detail.controller` will be instantiated and the constructor calls the `getProject` function. A project contains data from the JIRA project but also from the GitHub repository statistics. Accordingly, the `jira-data.service` and the `github-data.service` will be called. Both Angular services send HTTP requests to the JIRA and GitHub API, respectively. The API responses need to be transformed in the controller. For example, statistics about the project and repository have to be computed. These data transformations are being conducted by the function `transformProjectData` and `transformRepoData`. Subsequently, the transformed data needs to be merged into one project JSON object and finally can be displayed in the view.

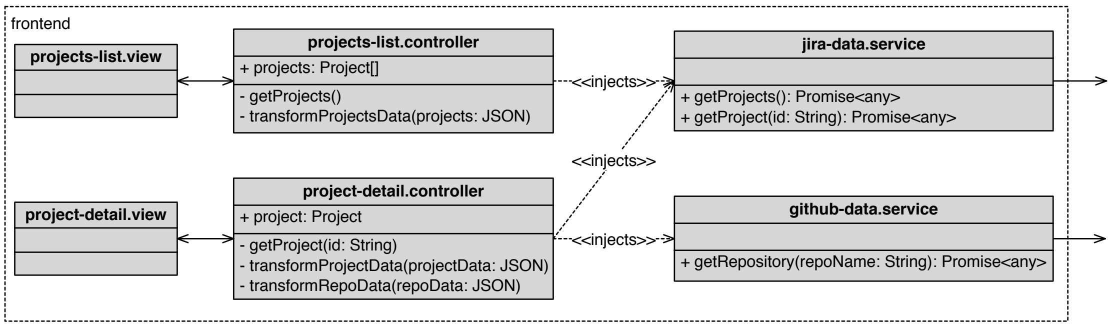

frontend
projects-list.view
projects-list.controller
+ projects: Project[]
- getProjects()
- transformProjectsData(projects: JSON)
<<injects>>
jira-data.service
+ getProjects(): Promise<any>
+ getProject(id: String): Promise<any>
<<injects>>
project-detail.view
project-detail.controller
+ project: Project
- getProject(id: String)
- transformProjectData(projectData: JSON)
- transformRepoData(repoData: JSON)
<<injects>>
github-data.service
+ getRepository(repoName: String): Promise<any>

Figure 5.5: Architecture of a conventional frontend contructed without a query service

5.3.2 Implementing RESTful API Consumption with the Query Service

Using the *query service* to handle RESTful API consumption leads to a client architecture represented by Figure 5.6. The frontend is also Angular based but additionally uses the Apollo framework in order to be able to communicate with the *query service* by sending GraphQL queries. The application obviously has the same views as the previously presented Angular app. Moreover, the two controllers, exchanging data with the view, are also represented in this architecture.
In such a frontend, just one data service is necessary since it communicates only with one API, namely the GraphQL API of the *query service*. The `project-data.service` consists of two functions: `getProjects` and `getProject`. Both functions send a query to the *query service* API and receive their data exactly as required by the view. They pass the response data as an observable to the controller. The controller only has to store the data in the variable (`projects` or `project`) which is bound to the view. The data transformations are conducted in the *query service* which makes further transformations in the client obsolete.

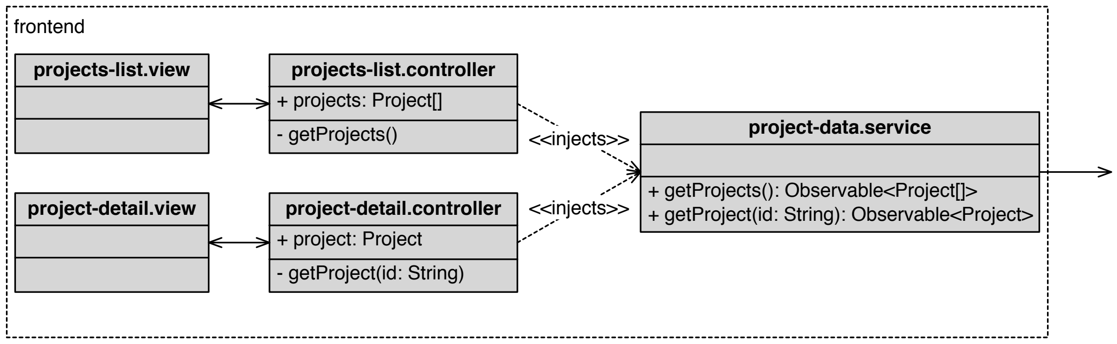


Figure 5.6: Architecture of a frontend using the *query service*

5.3.3 Comparing the Two Frontends

Comparing the two presented architectures reveals that the frontend, consuming the JIRA and GitHub API directly, contains more code. In this comparison only the frontend architectures are being analysed since the aim of the reference architecture is to ensure a lightweight frontend. As previously mentioned, the conventional Angular frontend is a matter of design and could be realised in a different way. For example, the data transformations could have been implemented in the data service instead of the controller. However, they have to be implemented somewhere in the code and this will still be on the client-side (since third party APIs are being consumed). The data services could also be realised as one service but this also would not reduce the amount of code.

Differences

The major differences of the two architectures are the data transformations in the controllers as well as the API consumptions in the data services. When utilising the *query service*, no data transformations are necessary on the client-side, since the data is being returned by the GraphQL API exactly as needed in the view. Furthermore, consuming the JIRA and GitHub API requires more code than sending two queries to a GraphQL server. The reason is that more than two HTTP requests need to be sent to the APIs in order to fetch all required data (the API requests that are necessary to conduct have been outlined in the previous sections).

Inference

The JIRA and GitHub APIs still need to be consumed when using a *query service* and the response data also needs to be transformed. However, this code is shifted away from the client (to the *query service*) leading to less complexity on the client-side. When

making use of the suggested reference architecture, the frontend developer does not need to think about how to fetch data from several APIs and how to implement the Angular services. In this way the entire focus can be on the user interface and how to present the data.
Nonetheless, the differences between the two architectures presented here are just marginal. This is due to the scope of the presented example scenario. When scaling up the number of APIs to consume, however, the amount of data service will increase as well as the data transformations. Thus, it might not be worth the effort using the query service for a small example. Nevertheless, the comparison has shown that the reference architecture does reduce the complexity in the client which is beneficial for a scenario consuming several RESTful APIs.

5.4 Advantages

This section evaluates the advantages of the presented approach based on the examples presented in the previous sections. It states why it is useful to utilise the web application when developing a web application. The benefits are related to using a model-driven approach and the reference architecture presented in this thesis.

5.4.1 Model-Driven Software Development

The code generation tool is driven by a model and based on MDSD. Hence, using the tool for the software development process of an application leads to advantages related to MDSD. In the following the main advantages of MDSD will be presented.

Software Quality The goal of MDSD is to enhance software quality. Automatic code generation leads to well structured and consistent code since it will always be generated precisely in the same way. Once an architecture has been defined as a model it will recur uniformly in the implementation [2].

Reusability and Portability The focus in MDSD lies on the development of PIMs which do not depend on a technology and therefore can be easily ported [54]. Reusability is also being enhanced since the code generation tool can be used over and over again for web applications. The goal of developing a model-driven tool is, therefore, to have code being used by several web applications instead of just one.

Development Speed The increase of the development speed can be achieved by automation [2]. Using code generation tools leads to less work for developers. Platform specific details do not need to be designed or written down [54]. The developers can focus entirely on constructing the PIM and will not be distracted by bugs or syntax errors (except when doing manual code refinement).

Manageability of Complexity Völter et al. [2] name as a further advantage of MDSD the manageability of complexity. This can be accomplished through abstraction. Focussing on the PIM allows the developer to concentrate on the problem itself and not on implementation details related to a technology. Thus, the complexity is reduced by programming on a more abstract level.

To sum up, the main benefits of such a code generation tool are the automation of the development process and the abstraction of the problem.

5.4.2 Leightweight Client

In section 5.3 a frontend using the *query service* for RESTful API consumption was compared to a frontend without using such a service. The comparison has shown that the reference architecture presented in section 3.3 does reduce complexity in the client. Accordingly, utilising the tool for web applications allows the frontend developer to focus on the design of the user interface and user experience. It shifts the responsibility of the data consumption outside of the client. This also counteracts the problem of having a monolithic frontend in a microservice architecture. Because of having to deal with all the services, the frontend might become more complex and hence the application cannot fully benefit from the advantages of microservices [32]. Shifting the API consumption outside of the client scales down the complexity in the frontend. This was the goal of establishing such a reference architecture. The idea was not to reduce complexity in general, but to have a less complex and more changeable client. Such a client is also a benefit when an API which is being consumed by the web application changes. Its consumption needs to be modified accordingly. Due to the presented reference architecture such API changes just need to be applied to the *query service*. The client does not need to be touched and therefore will stay fully functional.

5.4.3 Focus on Important Elements

The development process of the two example scenarios showed that using the tool allows quick access to APIs. As soon as you specify the URL and other requested parameters (header-, URI- or query parameters) of an API consumption you will receive working code to access the API. A developer would just have to read into the API specification of the service to access but not in the specification of a framework to access the APIs.
For an application like the one presented it is really convenient to use such a tool. Especially the second example application required a lot of thought in terms of the data model. It had to be considered where to get the data from as well as how to link it together. Therefore, reading into the API specifications is necessary to find out how to retrieve the values. However, using the tools allowed focussing on the data itself

and thinking about in what sequence the API requests need to be conducted. No time needed to be wasted on the implementation details as the semantics of a programming language. A developer does not have to think about how to realise the implementation of conducting an API request not before the other request completed. It is enough to specify the API request in the desired sequence. This is convenient, especially when being new to a programming language or framework. Hence, the tool allows putting the focus on the important parts of the development process.

5.4.4 Further Benefits

There are further benefits of using the code generation tool which did not emerge from the presented examples.

- The API of the generated *query service* can easily be tested. The server provides a user interface called GraphiQL[1] which is reachable by accessing the path `/graphiql` of the server. This allows trying out if the API response with the correct data after refining the code manually.
- Caching will be done automatically in the client. The *query service* uses the Apollo framework to establish a GraphQL server. Additionally, it is advisable to use the Apollo framework also in the client to facilitate the implementation of a GraphQL client. This framework is, for instance, used by the example implementation in the comparison of the two frontends. The Apollo framework has various features and one of them is caching. As stated by Apollo: *"...the tree structure of GraphQL lends itself extremely well to client-side caching"* [55]. The structure of a query or a mutation can easily be matched to the cache to see if the same request has already been conducted before. All requests are automatically cached by the Apollo framework. However, if it is desired not to use caching it can be overwritten by the `forceFetch` option [55].
- The code generation tool can be used offline. The tool itself does not need any server since data is not being stored in a data base. It just needs to load the JavaScript files once from a server and subsequently does not require any internet connection.
- The presented model describes the consumption of RESTful APIs and therefore affiliates in the description of web applications. There are several ways to describe different elements of a web application (e.g. data model, process model, view model) and the presented model now proposes a way to model API consumption.

[1]`https://github.com/graphql/graphiql`

5.5 Limitations

Certainly, the presented approach is not the silver bullet to the problem of the consumption of multiple APIs. It was not intended to present a solution that is better then all other approaches that already targeted the problem. The idea was to focus on different aspects, mainly the complexity reduction in the client. Clearly, this approach also has its drawbacks and use cases when it is not recommendable. Therefore, this section presents its limitations.

5.5.1 Downsides of MDSD

There are also limitations that come along with MDSD. In the previous section it was explained what benefits are related to automatic code generation. It allows having consistent code since it is being generated the exact same way every time using the tool. Therefore, it is extremely important to be very precise when developing such code generation tools. Implementing an error in the code that will be generated, can propagate this mistake in every application using this tool. As mentioned by Kleppe, Warmer and Bast [54] the payback for the effort of implementing model transformations is high but it needs to be carried out by highly skilled people.
One of the biggest limitations on the code generation tool is roundtrip engineering.

Roundtrip Engineering

Roundtrip engineering is about the synchronisation of the model and the generated code. It describes the possibility to make changes in both the model and the code and that the changes will always propagate bidirectional [2]. Forward engineering describes that if changes are made in the code they will also appear in the model. Reverse engineering describes the opposite, changes in the code will propagate to changes in the model. As stated by Völter et al. [2] this is a common issue in MDSD because it supports forward engineering but not reverse engineering. The reason for this is that in MDSD the model is more abstract than the code.
In the case of the presented model-driven tool the problem appears as follows: when constructing the model, the code generation process is influenced by the model and therefore represents the model. However, when now making changes in the code, the model will not be updated accordingly.
The problem even goes a bit further. Forward engineering is possible due to generation code from the model but since the model is not being stored in a database it will be lost once quitting the tool. Hence, the model is not permanently in synchronisation with the code. When, for example, one of the consumed APIs changes (e.g. when a new API version was released) the developer has two options: either adjust the code manually or specify a complete new model. In the latter case, though, code that was

added manually needs to be written all over again.
Even if the idea to solve this issue, would be to store the model in a database, further problems occur. When generating code from the model, subsequently manually refining this code and then adjusting the model again, what happens with the manually added code? How can this be stored in order not to loose it when generating code again from the updated model?

5.5.2 Level of Abstraction

The model presented in section 3.4 was described as a generic model. This is true when considering that it is a model for RESTful service consumption. However, it is not the highest level of abstraction since it is limited to the REST technology. In order to have a more abstract model it would be necessary to consider all possible ways of communicating with services. Even though REST is widely used on the internet, there are other concepts for APIs [56]. For example, an API using the Simple Object Access Protocol (SOAP) cannot be described with the presented model. Hence, the developer utilising the code generation tool is restricted to the consumption of APIs that are considered as RESTful.

5.5.3 Manual Code Refinement

In the section on roundtrip engineering it was already mentioned that manual code refinement can lead to problems when trying to counteract the issue. However, manual code refinement is generally a limitation on the code generation tool. Obviously, the tool would be easier to use if the code generation is 100% automatically. For a developer having to refine the code manually makes using the tool more complex. This step in the development process implies understanding the code that was being generated and being able to work with the used platform (NodeJS) and programming language (JavaScript). Mostly data just needs to be extracted from the API response by accessing a JSON object and returning that data. This should not be an obstacle for a web developer. But when having to do more complex transformations on the data and the developer is not very familiar with the language this might make the development process harder. It would be, therefore, desirable if the application developer could choose what platform of programming language to use for the generated *query service*. Thus, it can be ensured that the frontend developer understands the generated code much more easily.

5.5.4 Usability

Not only manual code refinement is a limitation on usability. The code generation tool is not an application that could be easily used by any user. It requires coding

knowledge especially in terms of HTTP request to RESTful APIs. The idea was to provide a tool for frontend developers who are familiar with such artefacts. However, even for experienced frontend developers using such a tool the first time requires a familiarisation process. The general idea of the application needs to be understood as well as how to construct the model. This can be a time consuming step in the process of application development and therefore it might not make sense to sue such a tool in all cases. Using the tool just once, for an application that does not need to consume lots of different APIs might not be worth the effort. In such a case manual coding could be quicker. Therefore, it is advisable to use the code generation tool when the data comes from several different APIs in order to profit from the advantages such as a quicker development process and complexity reduction in the client.
It should be noted that the tool was not truly evaluated in terms of usability in the two example cases. The development of the example applications was not conducted by a developer who never used the tool before. It is obviously easier to use a tool for someone who developed it than it would be for someone seeing it the first time.

6 Conclusion and Future Work

Contents

This final chapter concludes the present thesis. The conclusions will be listed which can be drawn on the basis of this work as well as the main limitation of the presented approach. Additionally, an outlook will be presented which outlines the future work.

6.1 Conclusion

The aim of this thesis has been to present a model-driven approach for the consumption of RESTful APIs in single page applications. Therefore, a reference architecture has been presented which shifts the consumption of RESTful APIs outside of the client. Additionally, a meta-model describing RESTful API consumption has been introduced as well as a code generation tool which is driven by this model. In order to achieve those contributions, related work and state-of-the-art methodologies and technologies have been analysed. Furthermore, the approach has been developed and evaluated in different ways in the scope of this thesis. Those procedures lead to the following conclusions that can be drawn from the results of this thesis:

- The consumption of RESTful services can be modelled. Analysing the state-of-the-art in web development technologies revealed the building blocks of RESTful API consumption. This led to the meta-model described in section 3.4 which acts as a basis for the code generation process.
- Consuming data from several different RESTful APIs leads to complexity in the client. This was found out from research. In single page applications more responsibility is shifted to the client. SPAs have to handle RESTful service consumption and related data transformations especially when data comes from third party APIs which do not provide the data in the desired way.
- The approach presented in the scope of this thesis reduces complexity in client. This has been shown in the evaluation by the comparison of two frontend architectures where one was developed with a *query service* and one without.

- Nonetheless, there are also limitations on the approach that have been revealed in this thesis. The biggest limitation on the tool is roundtrip engineering. The problem with roundtrip engineering is that with the presented code generation tool a *query service* can be generated, however, if the code will be changed, the model is not being updated accordingly. Furthermore, if it is desired to add or delete an API request, the model cannot be modified since it is not being stored. Hence, the code needs to be adjusted manually in such a case.

6.2 Future Work and Outlook

To keep this work in the scope of a master's thesis some restrictions have been made. Those are reflected in the limitations on the tool which have been outlined in section 5.5. Future work should address those limitations. Particularly, it is reasonable to focus on the following aspects:

Generic Model As mentioned in the limitations section, the meta-model only describes RESTful API consumption. Therefore, the code generation tool is only capable of generating code for RESTful API requests. It would make sense to go up one level in the abstraction and allow data exchange with any kind of server. This would result in a more generic model than the one presented in this thesis.

Store Model Another helpful feature for the code generation tool is the possibility to store the constructed model. This helps adjusting to the model if APIs change or have to be added. However, for such a feature it has to be preconceived how to store the manual code refinements with the model so that they are not lost when generating the server anew.

Real-Time API Management A further approach to solve the problem of adding and deleting APIs to the *query service* could be real-time API management. Such an approach is presented by Gadea et al. [57] who also propose a reference architecture for the consumption of RESTful APIs. They introduce an approach which allows real-time management of microservice APIs. They do not provide client-side code for the consumption of REST APIs but they allow adding and removing APIs during runtime. This could be a nice extension to the approach presented in this thesis.

Mapping Data Models The disadvantages that come along with manual code refinement have already been discussed. One way to avoid this problem could be a mapping process for the data models in the UI of the code generation tool. The user would specify in the UI how to map the data model from the response data of the RESTful API, to the data model of the client. However, integrating such a feature in the user interface seems not to be a trivial task.

User Testing In the evaluation the tool has been used to implement example applications. This step has been conducted by myself. Of course also research assistance were involved in the feedback of the code generation tool but no other users. Accordingly, it would be reasonable to perform user tests to evaluate and refine the tool in more depth. This could enhance the usability which is essential for third party users working with the tool. In general it would make sense to evaluate the tool and reference architecture in more depth by developing a bigger example application.

Security Adding security features to the *query service* is another aspect for future work. For example, OAuth could be added to the service which allows a secure connection between the client and the *query service*.

There are several more aspects that could be optimised, however, the main objective of modelling RESTful API consumption and generating the related code has been achieved.

Bibliography

[1] Mike Roberts. *Serverless Architectures*. 2016. URL: https://martinfowler.com/articles/serverless.html (visited on 01/08/2017).

[2] M Völter et al. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley Software Patterns Series. Wiley, 2013, pp. 3–28. ISBN: 9781118725764.

[3] Hamza Ed-Douibi et al. 'EMF-REST: Generation of RESTful APIs from Models'. In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (2015), pp. 1446–1453. DOI: 10.1145/2851613.2851782.

[4] Florian Haupt et al. 'A model-driven approach for REST compliant services'. In: *Proceedings - 2014 IEEE International Conference on Web Services, ICWS 2014* (2014), pp. 129–136. DOI: 10.1109/ICWS.2014.30.

[5] Rodrigo Bonifacio et al. 'NeoIDL: A Domain-Specific Language for Specifying REST Services'. In: *International Conference on Software Engineering and Knowledge Engineering* (2015), pp. 613–618. DOI: 10.18293/SEKE2015-218.

[6] Alan R Hevner et al. 'Design Science in Information Systems Research'. In: *MIS Quarterly* 28.1 (2004), pp. 75–105. ISSN: 02767783.

[7] Alan R Hevner. 'A Three Cycle View of Design Science Research'. In: *Scandinavian Journal of Information Systems* 19.2 (2007), pp. 87–92. ISSN: 09050167.

[8] Binildas A. Christudas, Malhar Barai and Vincenzo Caselli. *Service Oriented Architecture with Java*. Packt Publishing, 2008, pp. 33–35. ISBN: 978-1-847193-21-6.

[9] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, 2005, pp. 31–39.

[10] D Chappell. *Enterprise Service Bus*. O'Reilly Series. O'Reilly Media, Incorporated, 2004, pp. 1–21. ISBN: 9780596006754.

[11] Roy Thomas Fielding. 'Architectural Styles and the Design of Network-based Software Architectures'. PhD thesis. 2000.

[12] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, 2008, pp. 49–106.

[13] Tim Berners-Lee, Roy T. Fielding and Larry Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. STD 66. RFC Editor, 2005. URL: http://www.rfc-editor.org/rfc/rfc3986.txt.

[14] Dominik Renzel, Patrick Schlebusch and Ralf Klamma. 'Today's top "RESTful" services and why they are not RESTful'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7651 LNCS (2012), pp. 354–367. DOI: `10.1007/978-3-642-35063-4_26`.

[15] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. RFC Editor, 2014. URL: `http://www.rfc-editor.org/rfc/rfc7231.txt`.

[16] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. RFC Editor, 2014. URL: `http://www.rfc-editor.org/rfc/rfc7230.txt`.

[17] Efrem G. Mallach. *Information Systems*. CRC Press, 2015, pp. 122–125. ISBN: 978-1-4822-2374-3.

[18] Gil Fink and Ido Flatow. *Pro Single Page Application Development: Using Backbone.Js and ASP.NET*. Berkely, CA, USA: Apress, 2014, pp. 3–14.

[19] Martin Fowler and James Lewis. *Microservices - a definition of this new architectural term*. 2014. URL: `https://martinfowler.com/articles/microservices.html` (visited on 05/09/2017).

[20] Sam Newman. *Building Microservices*. O'Reilly Media, Incorporated, 2015, pp. 67–73. ISBN: 978-1-4919-5035-7.

[21] Dinkar Sitaram and Geetha Manjunat. *Moving To The Cloud*. Syngress, 2011, pp. 136–147. ISBN: 978-1-59749-726-8.

[22] Romanos Tsouroplis et al. 'Community-based API Builder to manage APIs and their connections with Cloud-based Services'. In: *Proceedings of the CAiSE 2015 Forum* (2015), pp. 17–23.

[23] SmartBear Software. *Swagger (OpenAPI) Specification*. 2016. URL: `http://swagger.io/specification/` (visited on 14/09/2017).

[24] RAML Workgroup. *RAML Version 1.0: RESTful API Modeling Language*. 2016. URL: `https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md` (visited on 14/09/2017).

[25] Atlassian. *Atlassian REST API Design Guidelines version 1*. URL: `https://developer.atlassian.com/docs/atlassian-platform-common-components/rest-api-development/atlassian-rest-api-design-guidelines-version-1{\#}AtlassianRESTAPIDesignGuidelinesversion1-RESTResources` (visited on 28/09/2017).

[26] Google. *API Design Guide*. URL: `https://cloud.google.com/apis/design/` (visited on 28/09/2017).

[27] Microsoft. *Microsoft REST API Guidelines*. URL: `https://github.com/Microsoft/api-guidelines/blob/master/Guidelines.md` (visited on 28/09/2017).

[28] Thomas Scheller and Eva Kuhn. 'Influencing factors on the usability of API classes and methods'. In: *Proceedings - 2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems, ECBS 2012* (2012), pp. 232–241. DOI: `10.1109/ECBS.2012.27`.

[29] Luis Augusto Weir. *Oracle API Management 12c Implementation*. Packt Publishing, 2015, pp. 4–8. ISBN: 978-1-78528-363-5.

[30] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002, pp. 466–472. ISBN: 978-0-321-12742-6.

[31] Michael S. Mikowski and Josh C. Powell. *Single Page Web Applications: JavaScript end-to-end*. Manning Publications, 2013. ISBN: 978-1-61729-075-6.

[32] Ruben Oostinga. *The monolithic frontend in the microservices architecture*. 2015. URL: `http://blog.xebia.com/the-monolithic-frontend-in-the-microservices-architecture/` (visited on 17/09/2017).

[33] Marco Brambilla, Jordi Cabot and Manuel Wimmer. *Model-driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012, pp. 9–10.

[34] Kevin Lano. *Model-Driven Software Development With UML and Java*. Boston, MA, United States: Course Technology Press, 2009, pp. 1–56. ISBN: 1844809528.

[35] Markus Lanthaler and Christian Gütl. 'Hydra: A vocabulary for hypermedia-driven web APIs'. In: *CEUR Workshop Proceedings* 996 (2013). ISSN: 16130073.

[36] Stefan Tilkov et al. *REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web*. 3rd ed. Dpunkt.Verlag, 2015.

[37] Davide Rossi. 'UML-based Model-Driven REST API Development'. In: *Proceedings of the 12th International Conference on Web Information Systems and Technologies, Vol 1 (WEBIST)* (2016), pp. 194–201. DOI: `10.5220/0005906001940201`.

[38] Dave Steinberg et al. *EMF: Eclipse Modeling Framework*. 2nd ed. Addison-Wesley Professional, 2008. ISBN: 978-0-321-33188-5.

[39] Areeb Alowisheq, David Millard and Thanassis Tiropanis. 'Resource Oriented Modelling: Describing Restful Web Services Using Collaboration Diagrams'. In: *e-Business (ICE-B), 2011 Proceedings of the International Conference on* (2011), pp. 1–6.

[40] Sandy Pérez et al. 'RESTful, resource-oriented architectures: A model-driven approach'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6724 LNCS (2011), pp. 282–294. ISSN: 03029743. DOI: `10.1007/978-3-642-24396-7_22`.

[41] Ruben Verborgh et al. 'Capturing the functionality of Web services with functional descriptions'. In: *Multimedia Tools and Applications* 64.2 (2013), pp. 365–387. ISSN: 13807501. DOI: `10.1007/s11042-012-1004-5`.

[42] Oliver Zeigermann and Nils Hartmann. *React: Die praktische Einführung in React, React Router und Redux*. Dpunkt.Verlag, 2016. ISBN: 978-3-86490-327-4.

[43] Kirupa Chinnathambi. *Learning React: A Hands-On Guide to Building Maintainable, High-Performing Web Application User Interfaces*. Addison-Wesley Professional, 2016. ISBN: 978-0-13-454631-5.

[44] Mithun Satheesh, Bruno Joseph D'mello and Jason Krol. *Web Development with MongoDB and NodeJS*. Packt Publishing, 2015, pp. 1–70. ISBN: 978-1-78528-752-7.

[45] Samer Buna. *Learning GraphQL and Relay*. Packt Publishing, 2016, pp. 6–88. ISBN: 978-1-78646-575-7.

[46] Sashko Stubailo. *GraphQL: The next generation of API design*. 2016. URL: `https://dev-blog.apollodata.com/graphql-the-next-generation-of-api-design-f24b1689756a` (visited on 21/09/2017).

[47] Mike Stowe. *Just Because Github Has a GraphQL API Doesn't Mean You Should Too*. 2016. URL: `https://www.programmableweb.com/news/just-because-github-has-graphql-api-doesnt-mean-you-should-too/analysis/2016/09/21` (visited on 21/09/2017).

[48] Roman Krictsov. *Moving existing API from REST to GraphQL*. 2016. URL: `https://medium.com/@raxwunter/moving-existing-api-from-rest-to-graphql-205bab22c184` (visited on 15/09/2017).

[49] Google. *AngularJS API Docs - $resource*. 2010. URL: `https://docs.angularjs.org/api/ngResource/service/{\$}resource` (visited on 08/09/2017).

[50] Facebook. *GraphQL Specification*. 2016. URL: `https://facebook.github.io/graphql/` (visited on 26/07/2017).

[51] StrongLoop Inc. *4.x-API - express()*. URL: `https://expressjs.com/de/api.html` (visited on 19/09/2017).

[52] Sinan Si Alhir. *Learning UML*. O'Reilly Media, Incorporated, 2003, pp. 129–152. ISBN: 978-0-596-00344-9.

[53] Nir Kaufman and Thierry Templier. *Angular 2 Components*. Packt Publishing, 2016, pp. 1–6. ISBN: 978-1-78588-234-0.

[54] Anneke Kleppe, Jos Warmer and Wim Bast. *MDA Explained: The Model Driven Architecture^{TM}: Practice and Promise*. Addison-Wesley Professional, 2003, pp. 7–10. ISBN: 978-0-321-19442-8.

[55] Dhaivat Pandya. *GraphQL Concepts Visualized*. 2016. URL: `https://dev-blog.apollodata.com/the-concepts-of-graphql-bc68bd819be3` (visited on 04/10/2017).

[56] Ole Lensmar. *Is REST losing its flair - REST API Alternatives*. 2013. URL: `https://www.programmableweb.com/news/rest-losing-its-flair-rest-api-alternatives/analysis/2013/12/19` (visited on 18/09/2017).

[57] Cristian Gadea et al. 'A Reference Architecture for Real-Time Microservice API Consumption'. In: *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms - CrossCloud '16* (2016), pp. 1–6.

Glossary

A

API Application Programming Interface
app application

B

BaaS Backend as a Service
BFFs backends for frontedends

C

CRUD create, read, update, delete

D

DSL Domain Specific Language

E

EMF Eclipse Modeling Framework
ESB Enterprise Service Bus

H

HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol

I

IDE Integrated Development Environment

J

Java EE Java Platform, Enterprise Edition
JCA Java EE Connector Architecture
JSON JavaScript Object Notation
JSX JavaScript Syntax Extension

M

MBE Model-Based Engineering
MDA Model-Driven Architecture
MDD Model-Driven Development
MDE Model-Driven Engineering
MDMA Model-Driven Model Architecture
MDSD Model-Driven Software Development
MVC Model View Control
MVVM Model View ViewModel

N

NPM Node Package Manager

O

OMG Object Management Group

P

PaaS Platform as a Service
PIM Platform-Independent Model
PSM Platform-Specific Model

R

RAML RESTful API Modeling Language
REST Representational State Transfer
ROA Resource Oriented Architecture

S

SOA Service Oriented Architecture
SOAP Simple Object Access Protocol

SPA Single Page Application

U

UI user interface
UML Unified Modeling Language
URI Uniform Resource Identifier
URL Uniform Resource Locator

W

WADL Wep Application Description Language

X

XMI XML Metadata Interchange
XML Extensible Markup Language

Appendix

A User Guide for the Code Generation Tool

This section presents a user guide for the code generation tool. It describes how to download and install the tool as well as its utilisation.

A.1 Prerequisites

Git

To clone the repository from GitHub, git is needed. It can be downloaded from:
`http://git-scm.com/`

Node

Node tools such as the node package manager (npm) are required to install and run the project. Node can be downloaded from:
`http://nodejs.org/`

A.2 Download

The implementation of the code generation tool is open source and can be downloaded from the following link:

`https://github.com/niklas92/service-modelling-tool`

The project can also be cloned by entering the following command in the console:

Listing 1: Command to clone the repository

```
git clone https://github.com/niklas92/service-modelling-tool.git
```

A.3 Installation

Before being able to build the project all packages, the implementation is dependent on, need to be install. Therefore, switch to the project directory and install the packages with npm:

Listing 2: Command to install dependencies

```
cd service-modelling-tool
npm install
```

After the dependencies have been installed, the project has to be build and started. For the build process the module bundler webpack[1] is being used. The following command is enough to build and run the application.

Listing 3: Command to build and start the app

```
npm start
```

Subsequently, the application can be accessed by directing to: `http://localhost:8000`.

A.4 Model Specification

Once, the application was started, the user can construct an instance of the model in the UI by entering data in the user form. In the following, all fields that can be specified in the form are listed and explained.

General Information

The first fields to fill out, specify general information about the *quer service* that is going to be generated. The fields *application name* as well as *port* are required.

[1]`https://webpack.github.io`

Table 1: Overview of the general information fields

Field Name	Expected Values	Description
Application Name	String	The name of the query service
Author Name	String	The name of the developer constructing the model
Port	Int	The port on which the query service will run for testing purposes (default is 8080)
Application Description	String	A description of the query service

Data Model

As a next step the data model of the client has to be specified. This is necessary so that the *query service* returns the requested data in the correct format to the client. Any number of entities can be defined here. The definition of one entity requires filling out the following fields:

Table 2: Overview of the data model fields

Field Name	Expected Values	Description
Entity Name	String	The name of the entity
Parameter Name	String	The name of the parameter which is contained in that entity
Parameter Type	String	The type of the parameter (checkout allowed types: subsection A.5)

Resolver

To specify where the data related to the specified data model comes from, resolver functions have to be determined. Any number of resolver functions can be specified. The fields *resolver name* and *return type* are required.

Table 3: Overview of the resolver function fields

Field Name	Expected Values	Description
Resolver Name	String	The name of the resolver function
Return Type	String	The type of the response of the resolver function (checkout allowed types: subsection A.5)
Argument Name	String	Name of an argument that can be passed to the resolver function
Argument Type	String	The type of the argument (checkout allowed types: subsection A.5)
Required	Boolean	Specifies if the argument has to passed along with the function or if it is optional

For each resolver function, several API request can be specified to determine where the data comes from that is being returned by the resolver. The *url* field is required.

Table 4: Overview of the API request fields

Field Name	Expected Values	Description
URL	String	The url of the API to consume. Including its URI parameters (surrounded by {})
HTTP Method	GET, POST, PUT, DELETE	The HTTP method used by this request
Request Body	JSON Object	body data that has to be sent along with the request
Authentication Username	String	The username for basic authentication with the API
Authentication Password	String	The password for basic authentication with the API
Parameter Name	String	The name of the parameter that will be passed along with the request
Parameter Value	String	The type of the parameter that will be passed along with the request (checkout allowed types: subsection A.5)
In	Header, Query	Specifying weather the parameter is passed along in the header or in the query of the request

A.5 Supported Types

In some fields of the tool's user form, parameter types have to be defined. According to Facebook [50] the accepted types are the following ones:

- `Int`: 32-bit integer
- `Float`: floating-point value
- `String`: UTF-8 char sequence
- `Boolean`: true or false
- `ID`: unique identifier
- Collections: In order to specify a collection of a certain type it has to be defined as an array of that type (e.g. `[Int]`).
- Custom types: If it is desired to use a custom type, it has to be defined as an entity in the data model of the *quer service* model.

A.6 Dependencies

Table 5: Dependencies of the *query service*

Name	Modules	Version	URL
Express	express	4.14.0	expressjs.com
Apollo	graphql	0.8.0	apollodata.com
	graphql-server-express	0.4.2	
	graphql-tools	0.8.0	

B User Guide for the Query Service

Once the model has been constructed through the UI of the tool, the code can be generated by clicking the *generate server* button. The *query service* will be generated automatically by the tool and downloaded as a zip file. This section explains how to run the *query service* and how the built-in testing interface works.

B.1 Installation

Firstly, the downloaded zip file has to be extracted. Subsequently, all packages the *query service* is dependent on, need to be installed. Therefore, execute the following command in the project directory:

Listing 4: Command to install dependencies

```
npm install
```

After the dependencies have been installed, the NodeJS server can be started with:

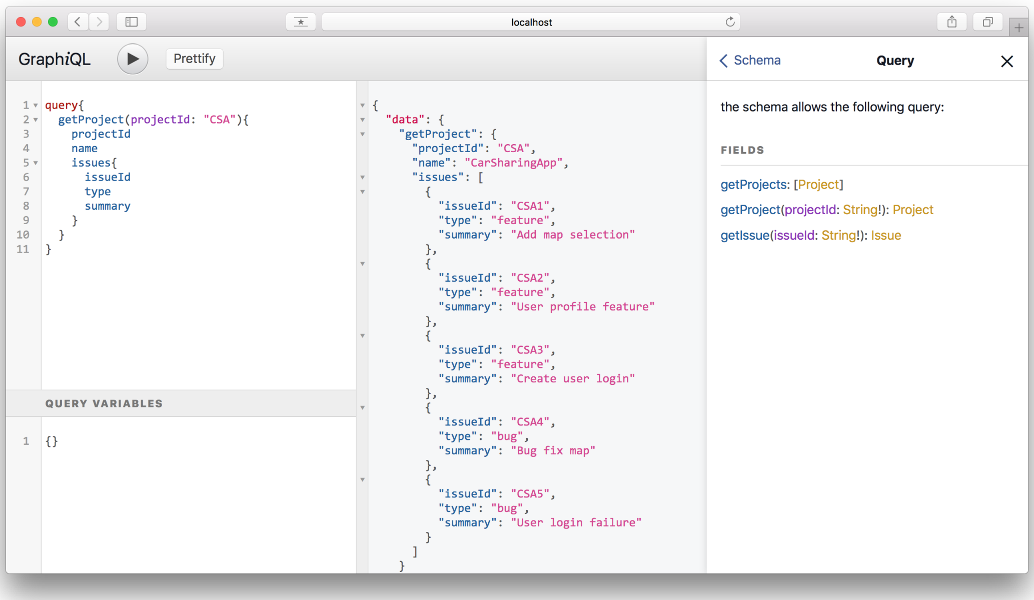
Listing 5: Command to start the query service

```
npm start
```

Subsequently, the *query service* will run on the specified port (8080 by default).

B.2 Testing

Once the *query service* is running, it can be tested with the help of GraphiQL. GraphiQL is a testing framework which allows sending queries and mutations to the GraphQL API of the *query service*. It can be accessed by directing to `http://localhost:8080/graphiql` (or a different port if the default port has been changed when constructing the model). Figure 1 shows a screenshot of the GraphiQL UI. On the left hand side test requests can be entered. Queries or mutations can be inserted in the upper field. Related query variables can be entered in the lower field. After sending the request to the server by pressing the *"play"* button, the response will be displayed in the middle view. The schema can be explored on the right hand side of the GraphiQL interface. There, all types, queries, and mutations are listed.



```
query{
  getProject(projectId: "CSA"){
    projectId
    name
    issues{
      issueId
      type
      summary
    }
  }
}
```

```
{}
```

```
{
  "data": {
    "getProject": {
      "projectId": "CSA",
      "name": "CarSharingApp",
      "issues": [
        {
          "issueId": "CSA1",
          "type": "feature",
          "summary": "Add map selection"
        },
        {
          "issueId": "CSA2",
          "type": "feature",
          "summary": "User profile feature"
        },
        {
          "issueId": "CSA3",
          "type": "feature",
          "summary": "Create user login"
        },
        {
          "issueId": "CSA4",
          "type": "bug",
          "summary": "Bug fix map"
        },
        {
          "issueId": "CSA5",
          "type": "bug",
          "summary": "User login failure"
        }
      ]
    }
```

the schema allows the following query:

getProjects: [Project]

getProject(projectId: String!): Project

getIssue(issueId: String!): Issue

Figure 1: Screenshot of the testing interface GraphiQL

B.3 Dependencies

Table 6: Dependencies of the code generation tool

Name	Modules	Version	URL
React	react react-dom react-tap-event-plugin	15.6.1 15.6.1 2.0.1	reactjs.org
Webpack	webpack	3.3.0	webpack.github.io
Mustache	mustache mustache-loader	2.3.0 1.0.0	mustache.github.io
Material UI	material-ui	0.18.7	material-ui.com
JSZip	jszip	3.1.3	stuk.github.io/jszip/
FileSaver	file-saver	1.3.3	npmjs.com/package/file-saver