A Model-driven Approach for Generating RESTful Web Services in Single-Page Applications

Adrian Hernandez-Mendez, Niklas Scholz and Florian Matthes
Technical University of Munich, Department of Informatics, Munich, Germany

Keywords: Model-driven Software Engineering, Web Services, Single-Page Application, Resource Oriented Architecture, RESTful APIs.

Abstract: Modern Single-Page Applications (SPA) use data from multiple Web services to support essential process in the enterprises. By using data from several Web services, the SPA changed their architecture from a one-to-one communication between client and server to an application using information from multiple servers using RESTfuls APIs in a microservice architecture. In this paper, we present a model-driven approach for the consumption of RESTful Web services in SPA. We introduce a Query Service meta-model and provide a tool to semi-automatically generate an SPA based on our reference architecture. The proposed approach was evaluated by using the tool for the development of an example application in the context of a research project with large German corporation in the domain of software architecture. The main limitation of the tool is the lack of support for the round-trip engineering functionality. However, the created Web service handles the access to APIs and reduces the complexity of the SPA due to the shift of responsibility away from the client.

1 INTRODUCTION

Nowadays, Single-Page Applications (SPA) support relevant processes in the enterprises, and they are not limited to show static information to the users. Additionally, their architecture has changed from a one-to-one communication between client and server to a client using information from multiple servers using RESTfuls APIs in a microservice architecture. Examples can be seen in digital companies such as Netflix and Airbnb, where their websites are supported by the integration and collaboration between several services.

Using several services in an SPA introduces new challenges in the implementation. Particularly, when a service was designed by a third-party, which implies that the provided data does not entirely fit the SPA needs. Consequently, the data has to be transformed on the client side. This data transformation process leads to increase the complexity of the SPA, especially when scaling up the number of services to consume, which could lead to compromises the SPA agility and changeability.

We envision an approach to reduce the SPA architecture's complexity by shifting the responsibility of managing the integration of multiple services from the SPA to a single RESTful Web service. Thus, the SPA just requires managing one service to receive all the needed information. This service provides the data exactly as needed by the client so that no further transformation is necessary and the developer obtains a lightweight SPA.

In this paper, we introduce a model-driven approach to automate the implementation of the consumption of RESTful Web services. We, therefore, develop a model which describes the required artifacts to consume RESTful services in single page applications. Subsequently, we present a transformation process where an application developer has to specify the transformation rules. The result of this process is a generated service that handles the RESTful Web service consumption for the client.

We have adopted a design science research approach, for the conceptualization of the model-driven approach as a design artifact. According to Hevner's three-cycle view of design science Hevner (2007), our research contributes in the following ways to each of the cycles.

Relevance Cycle: We stimulate discussions regarding complexity in the development of SPA with our industry partners in the context of a research project with large German corporation in the domain of software architecture. Thereby, we establish the need for a formal process for analyzing and extending the service

Hernandez-Mendez, A., Scholz, N. and Matthes, F.
A Model-driven Approach for Generating RESTful Web Services in Single-Page Applications.
DOI: 10.5220/0006608204800487
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 480-487
ISBN: 978-989-758-283-7

model based on existing SPA prototypes.
Design Cycle: We present the conceptualization of the model-driven approach, a query service model, and a reference architecture as our design artifacts.
Rigor Cycle: We have studied and evaluated the literature on existing approaches to model the RESTful Web services. In addition to the proposed model and reference architecture, we contribute to the knowledge space by providing a theoretical framework to discuss with industry partners and the research community in the MDD area.

2 RELATED WORK

Model-Driven Software Development (MDSD) has proven to be a good approach for managing complexity by enhancing the level of abstraction Völter et al. (2013). It also improves the software quality and increases the development speed. In MDSD, the development process is driven by formal models which will then be automatically transformed to code. The starting point is a platform independent model (PIM) which models the system regarding domain concepts and is independent of an implementation Lano (2009). This way, the developer can focus on specifying where the data for the client comes from and does not get carried away with the technical details of the service consumption.
Interesting solutions have been published in the field of MDSD which also focussed on Web services. Several approaches deal with modelling RESTful Web services services and transform them to code (Haupt et al. (2014), Ed-Douibi et al. (2015), Bonifacio et al. (2015) and Rossi (2016)). However, the focus of these approaches is generating the services on the server instead of the user of them by the clients (i.e., SPA). Our approach, on the contrary, focuses on the client-side by presenting a model-driven approach dealing with the consumption of RESTful Web services.
The challenge of managing multiple web services consumed by one application is not relatively new and has already been addressed by several other approaches. For example, the Service Oriented Architecture (SOA) is addressing this challenge by providing an enterprise service bus (ESB). The ESB acts as the backbone of the application and allows integration and management of services in an application (Chappell (2004)). Also API management tools such as apigee[1] or WSO2[2] support the usage of multiple APIs. Both also provide a gateway for APIs (similar to our *Query Service*), allow API management, as well as several other functionalities. These approaches, however, focus on the actual management of existing RESTful APIs, whereas our approach targets the development of SPA.
Additionally, in Gadea et al. (2016) an approach allows real-time management of microservice APIs is presented and a reference architecture for the consumption of RESTful APIs. This approach allows adding and removing APIs during runtime. However, a client-side code for the consumption of REST APIs is not considered in the approach.

[1]https://apigee.com/api-management/
[2]http://wso2.com

3 MODELING THE RESTFUL WEB SERVICES

Consuming data from several different APIs leads to complexity in the client. To counteract this complexity, our approach proposes to shift the responsibility of the API consumption outside of the client. We introduce a *Query Service* which handles requests to RESTful Web services and the subsequent data transformation. This *Query Service* provides one single interface for the client and supplies the data exactly as needed by the client. Therefore, data transformation operation in the SPA is not further required. 1 gives an overview of such an architecture.

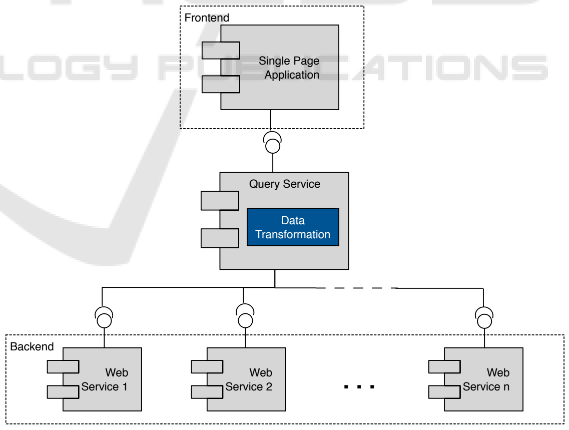


Figure 1: Architectural overview of a web application using the *Query Service*.

The *Query Service* is an instance which runs separately from the client and can be regarded as a web service. 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.

3.1 The Query Service Model

The *Query Service* meta-model reveals what information is needed to conduct CRUD operations on RESTful APIs. To establish this meta-model, the client-side applications and how they handle their data transfer with the backend were inspected and evaluated. For example, in Angular[3] we evaluated the current implementation of the Angular services.
2 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 also needs to specify a *data model*, this is the data model of the frontend, and 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. 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* and by potentially passing *arguments* along. Since the *Query Service* just manages access to several different RESTful Web services a *resolver* also needs an *API request*. 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.). Where applicable, *header parameters*, a *body* and/or *authentication* need to be passed along with the request.

4 MODEL-DRIVEN APPROACH FOR GENERATING RESTFUL WEB SERVICES

To develop the RESTful Web Services, we propose a semi-automatic process which is driven by the meta-model described in 3. 3 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 platform independent model (PIM), (2) transformation of the PIM to a platform specific model (PSM) to support the format of a GraphQL[4] server, (3) code generation based on the PSM, and (4) manual code refinement by the developer.

[3]https://angular.io

[4]http://graphql.org

4.1 Design Decisions

To provide a practical implementation of the architecture presented in 3, we provide a semi-automatically generated *Query Service*. We chose to implement it as a GraphQL server. GraphQL is a query language for APIs. This allows the client to send much more powerful queries to the API then it would be possible with an ordinary REST request. The client can exactly specify the structure of a data entity it wants to be returned. Like this, 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.
As the server we chose NodeJS[5] since it is convenient to configure and ensures a lightweight server. To make this GraphQL-based server, we used Apollo[6], which is a framework for implementing a GraphQL server.

4.2 Model Construction

The generation process of the *Query Service* starts with the model construction. 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 3 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 to resolve the data. After the developer finished constructing an instance of the model it is stored in a JSON object for further process of the application.

4.3 Model Transformer

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 Völter et al. (2013). In our case, this platform is the GraphQL server. The model which was constructed in the previous step, thus, needs to be transformed to a certain structure to be able to generate the platform afterward. A GraphQL server is specified by defining a *schema* and *resolvers*. The GraphQL *schema* defines which queries can be sent by the client, hence, describes the API of the GraphQL server. It specifies the data types, functions to get data (so-called *queries*) and functions to alter data (so-called *mutations*). The

[5]https://nodejs.org/

[6]https://www.apollodata.com

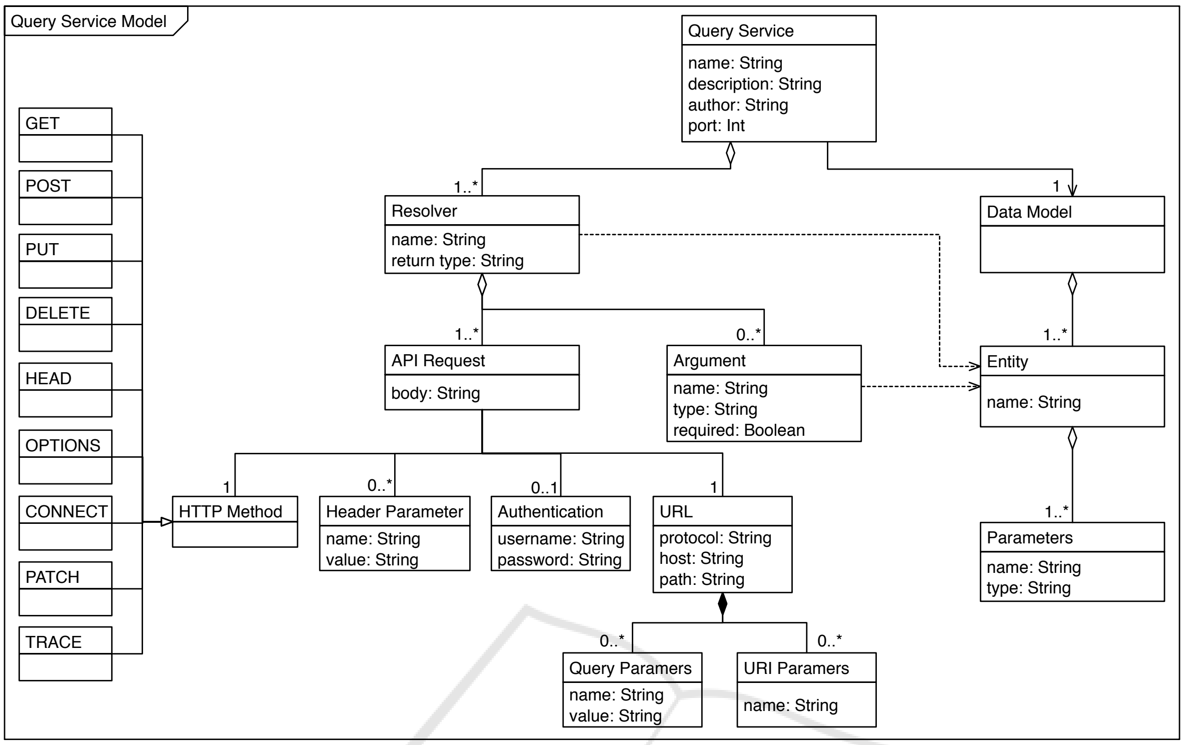


Figure 2: Meta-model of the *Query Service*.

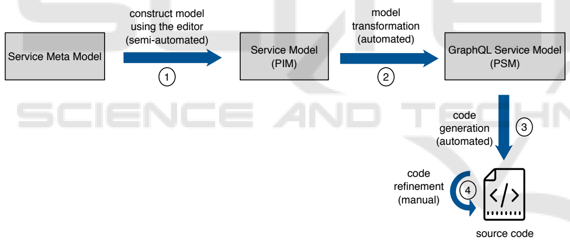

Service Meta Model
construct model using the editor (semi-automated)
1
Service Model (PIM)
model transformation (automated)
2
GraphQL Service Model (PSM)
code generation (automated)
3
code refinement (manual)
4
source code

Figure 3: Overview of the model transformation process.

GraphQL *resolvers* determine the data transfer for each one of the *queries* and *mutations*.
Accordingly, we need to construct this *schema* and the *resolvers* in the format as specified by GraphQL Facebook (2016). 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. The definition of the data types for the GraphQL *schema* can be extracted from the data model. The queries and mutations definition in the *schema* can be derived from *resolver* (name and return type), *argument* and *HTTP method* (reveals if it has to be defined as a query when it is a GET method or as a mutation otherwise). Information for GraphQL *resolvers* is extracted from everything related to the *resolver* in the *Query Service* model.
Since we are using a NodeJS runtime, we realise the API requests as `http.request`[7] and `https.request`[8] functions, respectively.

4.4 Renderer

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[9]. The mustache templates contain the static content, meaning the code that needs to be 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

[7]https://nodejs.org/api/http.html
[8]https://nodejs.org/api/https.html
[9]https://mustache.github.io

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.

4.5 Code Refinement

In MDSD 100% code generation is only for exceptional cases possible (Völter et al. (2013)). Thus, such a process almost always includes a step carried out manually by the application developer.
In our process, this 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 Web services. Accordingly, the received data needs to be mapped to the data model expected by the client.
It was not possible for us to model this data transformation after 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 part of the received data belongs to what part 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.

4.6 IMPLEMENTATION & ARCHITECTURE

The *Query Service* creator tool is a React[10] web application which implements the code generation described in the previous section. An overview of the architecture of the web application is represented by 4.

[10]https://facebook.github.io/react/

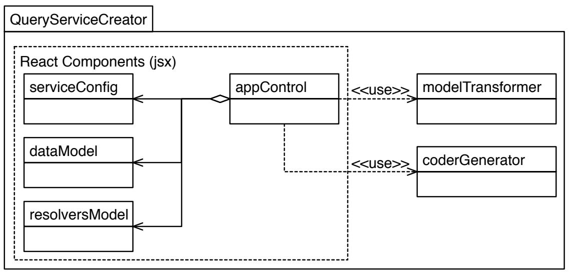


Figure 4: Architecture of the *Query Service* creator tool.

The *appControl* component is the center of the application and directs the program flow. The user interface is divided into three elements of the *Query Service* model: the general service information, the data model, and the resolvers. Those three elements are represented by the react components: *serviceConfig*, *dataModel*, and *resolversModel*. Each one of these components consists of a web form which is being filled out by the application developer to construct the relating part of the model.
5 displays the dynamic behavior of the tool. After the developer submits the model instance, the *appControl* component passes it along to the *modelTransformer* to transform it suitable for the GraphQL server by constructing the schema and resolver functions.
Once the model is in the correct format, the files can be rendered by the *codeGenerator*. Therefore, the *appControl* passes the JSON object constructed by the *modelTransformer* which holds the model information 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.

5 EXAMPLE SCENARIO

In this section, we evaluate our approach by presenting an example web application which is being implemented with the help of our tool. Firstly, we explain the functionality of the web application to develop, followed by a description of the development process realized with our tool. We conclude this section by analyzing the development process and stating the limitations on the code generation tool.

5.1 Scenario

The main functionality of this example web application is to provide statistics about completed software projects related to an organization and make predictions based on this data for future projects.

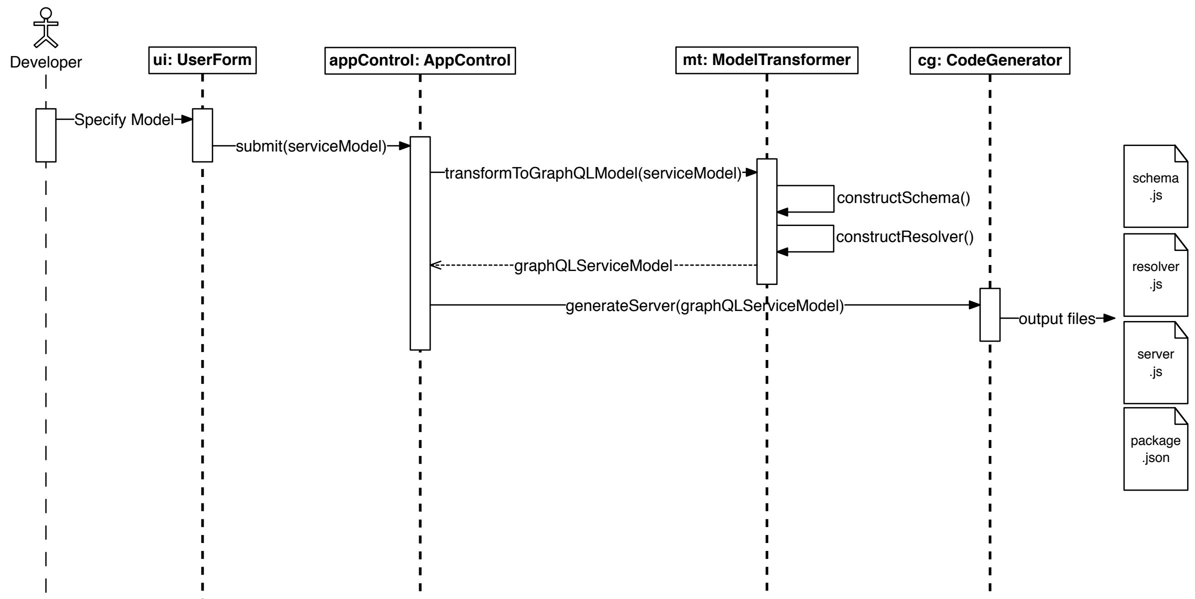
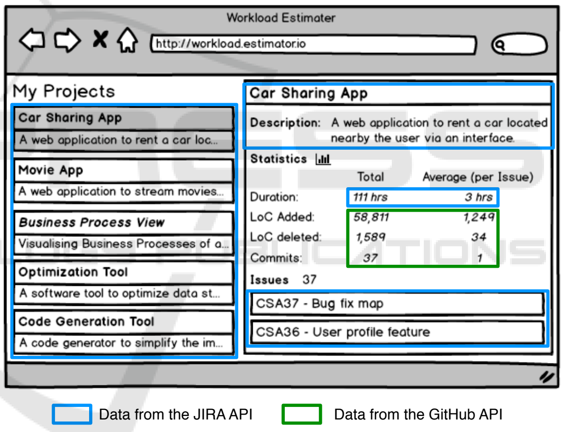


Figure 5: Sequence diagram of the *Query Service* creator tool.

6 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 to 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 organization utilizes JIRA as a project management tool. Code related data is being provided by the GitHub API assuming the organization uses GitHub as a project repository.
For the sake of evaluating our code generation tool we just consider the part of the development process which relates to the consumption of the JIRA and GitHub API. Other development steps such as the user interface implementation are not relevant for us.

5.2 Development Process

We start the implementation process by constructing an instance of the *Query Service* model with the help of the tool presented in 4.


Workload Estimater

http://workload.estimatorio

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

Figure 6: Mockup of the example web application and where its data comes from.

5.2.1 Data Model

Initially, we enter the general information about the web service we are going to generate (name, author, etc.). Subsequently, we have to think about the data model for the client side. To present the data in a web application as stated in the previous section, we come up with two entities: *Project* and *Issue* (cf. 7).
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.

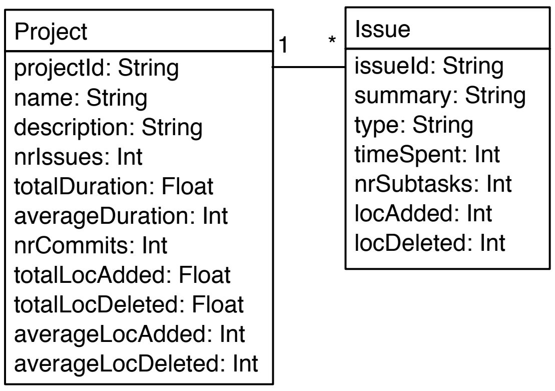


Figure 7: Data model of the example web application.

5.2.2 Resolver

As a next step, we define the resolver functions to specify from what APIs the data for the entities comes from. Three resolver functions are necessary: `getProjects` to return a list of all software projects, `getProject` to return data of one specific project including its issues, and `getIssue` to retrieve detailed information related to that issue. We are not explaining all three resolver functions but present the specification of `getProject` as an example.
Firstly, we enter the resolver name (*getProject*) and the return type (*Project*) and specify that we need the *projectId* as an argument for the function to be able to retrieve the correct project. Since we are using the JIRA project key as our id, the type is a *String*. Subsequently, we specify the APIs to consume to fetch the data related to the project.

JIRA API We need to access the JIRA API to retrieve the general information about the project as well as the issues related to it. As a query parameter, we pass along the projectId (the function argument). We also need to add authentication (username and password) to be able to access the data.

GitHub API We use the GitHub API to get statistics about the project's repository. In the URL we pass the name and owner of the repository along as URI parameters.

The other two resolver functions need to be specified accordingly. The code can be generated when all the required fields are complete.

5.2.3 Manual Code Refinement

To finish the development of the *Query Service*, the generated code needs to be refined manually. Therefore, we need to add code in the resolver functions to specify what we want to happen after each API request completed. We extract the data that is needed in the client from each API response and return it. In some cases, we also need to do some calculations such as compute the average number of lines of code added per issue.
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 transformation since it is being returned exactly as needed.

5.3 Critical Reflexion

5.3.1 Advantages

Using the code generation tool for the software development process of an application provides the standard advantages related to MDSD. For example, software quality is being increased since code generation automatically leads to well structured and consistent code. Other advantages that come along with MDSD are the enhancement of development speed, a higher level of reusability and the improved manageability of complexity (Völter et al. (2013)).
Additionally, the development process 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 the working code to access the API. Using the tool allows putting the focus on the important parts of the development process and not the semantics of a programming language.
If we would develop the example web application as an AngularJS app and not use our tool, it would result in a more complex client. To realize this example project in AngularJS, we would have to implement the resolvers as Angular services and the API requests with `ngResource`. We would have several API requests for each service and also implement the data transformation processes in the client. However, if you use the architectural approach, we propose all this code is outside of the client, and the frontend developer does not need to think about how to fetch data but can concentrate on the user interface.

5.3.2 Limitations

Our approach requires an initial familiarisation with the model and process. Therefore, using the tool just once, for an application that does not need to consume several services might not be the best scenario. In such a case, manual coding could be quicker. Thus, it would make sense to improve the usability of the code generation tool to counteract this problem.
However, the main limitation of our approach is the

lack of round-trip engineering functionality. Once the model is specified, and the code is generated, the model and the code are not in synchronization anymore. 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 completely new model. In the latter case, though, code that was added manually needs to be written all over again.
The meta-model we presented is not entirely generic. It is limited to RESTful APIs and therefore, cannot be applied with APIs using a different standard such as SOAP.

6 CONCLUSION

In this paper, we presented a model-driven approach for the consumption of RESTful APIs in SPA. We introduced a reference architecture that reduces the complexity of the SPA when using multiple different APIs. We provided a meta-model describing the consumption of RESTful APIs. Based on this meta-model a code generation tool was developed to create a Web service. We evaluated by utilizing the code generation tool for the development of an example application. The created Web service handles the access to APIs and reduces the complexity of the SPA due to the shift of responsibility away from the client.
Our model is already designed to be a generic model. However, we limited the API consumption to REST. As future work, we want to go up one level in abstraction and allow data exchange with other technologies. Another part to focus on is the support of the round-trip engineering functionality, which is currently the main limitation of our tool. For example, the architecture presented by Gadea et al. could be a nice extension to our approach to counteract this problem.

REFERENCES

Bonifacio, R., Castro, T. M., Fernandes, R., Palmeira, A., and Kulesza, U. (2015). NeoIDL: A Domain-Specific Language for Specifying REST Services. *Seke*, pages 613–618.

Chappell, D. (2004). *Enterprise Service Bus*. O'Reilly Series. O'Reilly Media, Incorporated.

Ed-Douibi, H., Izquierdo, J. L. C., Gómez, A., Tisi, M., and Cabot, J. (2015). EMF-REST Generation of RESTful APIs from Models. *CoRR*, abs/1504.0:39–43.

Facebook (2016). GraphQL Specification. https://facebook.github.io/graphql/ [Accessed: 26/07/2017].

Gadea, C., Trifan, M., Ionescu, D., and Ionescu, B. (2016). A Reference Architecture for Real-Time Microservice API Consumption. In *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms - CrossCloud '16*, pages 1–6, New York, NY, USA. ACM.

Haupt, F., Karastoyanova, D., Leymann, F., and Schroth, B. (2014). A model-driven approach for REST compliant services. In *Proceedings - 2014 IEEE International Conference on Web Services, ICWS 2014*, pages 129–136.

Hevner, A. R. (2007). A three cycle view of design science research. *Scandinavian journal of information systems*, 19(2):4.

Lano, K. (2009). *Model-Driven Software Development With UML and Java*. Course Technology Press, Boston, MA, United States.

Rossi, D. (2016). UML-based Model-Driven REST API Development. In *Proceedings of the 12th International Conference on Web Information Systems and Technologies, Vol 1 (WEBIST)*, pages 194–201.

Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S., Czarnecki, K., and von Stockfleth, B. (2013). *Model-Driven Software Development: Technology, Engineering, Management*. Wiley Software Patterns Series. Wiley.