Model-driven Development of RESTful APIs

Vitaliy Schreibmann and Peter Braun

Faculty of Computer Science, University of Applied Science Würzburg-Schweinfurt, Sanderheinrichsleitenweg 20, Würzburg, Germany

Keywords: REST, RESTful, API, Model-driven Software Development.

Abstract: We propose a model-driven approach for creating RESTful APIs. Today, REST APIs are developed by using frameworks and libraries that support software developers on the level of the chosen programming language, i.e., a lower level of abstraction. We argue that the development process can benefit from a model-driven approach, where an API is modeled on a higher level of abstraction by using a new formal language that was designed in particular for this application domain. From this model, the source code of the API is generated automatically, along with necessary code on the business logic and database layer. The benefits of this approach include higher productivity, better maintenance, higher quality, and documentation for free.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

In the year 2000 Roy Fielding proposed as part of his thesis (Fielding, 2000) a new architectural style for distributed systems, called Representational State Transfer (REST). It comprises of several high-level principles on how to design a distributed system. For example, according to the client/server principle, a communication act is always initiated by clients that send requests to servers. Another principle is stateless communication, which means that the server is not aware of any application state of one specific client. The last principle we mention here is hypermedia, which recommends to let the server drive clients through the states of an application by using hyperlinks.

Although REST influences the software architecture of a complete distributed system (Costa et al., 2014; Fielding and Taylor, 2002), it is mostly reduced to the Application Programming Interface (API) between clients and servers, where REST must be seen in contrast to other techniques for communication in distributed systems, for example Remote Procedure Calls (RPC) (Coulouris et al., 2005) or SOAP (Papazoglou, 2008). For a comparison of SOAP and REST, see (Pautasso et al., 2008). In the community, the term RESTful is used to express that an API complies with the major principles of REST, although this term has not been defined exactly so far (Richardson and Ruby, 2007). Ongoing research by (Klein and Namjoshi, 2011) attempts to formally prove RESTful behavior for the two constraints hypermedia and stateless communication.

The main advantage of REST is its simplicity in using an API, compared to other approaches such as SOAP. This is one reason REST interfaces are becoming increasingly popular for developers of Web-based and mobile applications. Other reasons are the clear intentions of the API and guidance through hypermedia for API users. In contrast, the design and implementation of REST interfaces on the server-side is rather difficult, time-consuming, and error-prone, because there is a huge conceptual gap between the ideas of REST on the level of Fielding's thesis and their implementation. In our opinion, the reasons for this are twofold:

1. Since REST is an architectural style, it is open for interpretation. For a developer it is simple to select only few principles of REST but missing the decisive ones. For example, many existing APIs are called RESTful only because they are built on the concept of resources and proper HTTP verbs but neglect the hypermedia principle (Fielding, 2008).
2. Developers lack a comprehensive tool-chain for design, documentation, and testing REST interfaces. Available frameworks and libraries on the level of programming languages only focus on a low level of implementation. For example, they support dispatching of incoming requests to an implementation based on the URI in an easy way

Schreibmann V. and Braun P..
Model-driven Development of RESTful APIs.
DOI: 10.5220/0005411200050014
In *Proceedings of the 11th International Conference on Web Information Systems and Technologies* (WEBIST-2015), pages 5-14
ISBN: 978-989-758-106-9
Copyright © 2015 SCITEPRESS (Science and Technology Publications, Lda.)

and allow to read and define HTTP header information. These tools do not ensure implementing a RESTful interface, because they only simplify the usage of the HTTP protocol but do not enforce Fielding's constraints.

This paper addresses the problem of providing an API according to the principles of REST in an easy way. Our goal is to bridge the gap between high-level concepts of REST and the low-level of implementation of an interface in a specific programming language. For this, we use a model-driven approach, in which the technical domain is limited to a sub-set of RESTful interfaces. We will argue that this sub-set is sufficient for most use cases. Model-driven software development (MDSD) is a technique to generate source code from an abstract model (Stahl et al., 2006). Instead of using a Turing-complete programming language with statements, conditions, and loops, MDSD enables developers to think on a higher level of abstraction. In our case, developers have to deal with basic concepts of RESTful APIs, such a resources, application states, and caching strategies only to name a few. The model is defined by using a textual domain specific language (DSL), named RDSL. From this model, a software generator creates a deployable RESTful interface.

Using such a model-driven approach not only reduces the complexity during development of RESTful interfaces but also improves quality assurance by generating automated test cases and simplifies usage of this API by generating libraries for accessing it from different types of clients.

The rest of this paper is structured as follows. In Chapter 2 we discuss other approaches to facilitate the development of RESTful APIs. Chapter 3 explains why it is reasonable to further restrict the flexibility of developers without necessarily narrow the number of use cases. In Chapter 4 we introduce a meta-model for RESTful APIs and in Chapter 5 we show examples of RDSL. Finally, in Chapter 6 we discuss our approach and give an outlook to further development.

2 RELATED WORK

Frameworks and libraries aim at supporting developers in creating RESTful APIs on the level of programming languages. For example, the Dropwizard framework (Dallas, 2014) is built on top of already existing tools and libraries such as Jetty, Jersey, Jackson, and Hibernate to provide a ready-to-be-used toolbox for backends and clients. These tools and frameworks support developers in writing a RESTful APIs but do not enforce it. Unfortunately, developers are required to have programming skills and a thorough understanding of the principles of REST in order to develop a proper interface.

Several attempts were made to generate RESTful services, but these approaches only focus on resources and methods to create, read, update, and delete (CRUD operations) these resources. (Pérez et al., 2011; Schreier, 2011) have developed their own REST meta-models using Amazon S3 as reference API, overlooking the importance of the hypermedia constraint.

Apimatic[1] and RAML[2] try to covers all aspects concerning REST from API definition on the backend to generation of SDKs for frontends. With Apimatic the developer has to interactively define resources and their attributes using a Web-interface. Apimatic then generates an API providing CRUD operations on these resources and SDKs to be used in clients on Android and iOS. RAML is the first high-level language for developers to define, create, test, and publish RESTful APIs. RAML uses YAML as markup language and is based on the idea of defining resources and their representations as JSON schemas. Currently, neither Apimatic nor RAML support the concept of hypermedia but hypermedia is essential to increase flexibility and comparability of an API. In addition, with properly applied hypermedia, clients become flexible, stable against changes, and could evolve beyond pure data consumers.

(Lanthaler and Gutl, 2010) compared different Web service description languages including WSDL and WADL. WSDL 2.0 is able to describe REST based services but in our opinion they increase complexity against the straightforward principles of REST. WSDL 1.0 and WADL could also formally describe REST but not RESTful APIs due to the lack of hypermedia support in the specification. As a consequence, the research of (Tavares and Vale, 2013) omitted hypermedia in their meta-model and in the transformation process to WADL. Similar WADL descriptions were produced by (Laitkorpi et al., 2009) including the hypermedia constraint in their model-to-model transformation. Replacing WADL with RDSL simplifies the definition of REST APIs for users and RDSL could define features WADL never could such security, resource states, API documentation, and testing

We decided to omit the support for legacy systems in favor of straightforward creation of complete and deployable REST APIs, which should allow to replace any legacy system in the future. In the first iteration of this project the focus is on structural mod-

[1] https://apimatic.io/
[2] http://raml.org/

els with the intention to extend it with a behavioral model similar to the work of (Porres and Rauf, 2011).

3 TOWARDS REASONABLE CONSTRAINTS FOR RESTful APIs

Currently, the development of a RESTful API is based on frameworks and libraries. For example, JSR 339[3] specifies Java annotations to map incoming HTTP requests to specific classes and methods. The annotation `@GET` in combination with an annotation `@PATH("/orders")` placed at a method defines that this method should be called in case of a HTTP GET request to the URI with `/orders` as last path element. However, this approach does not prevent developers to wrongly use GET requests for updating resources, although it violates the safety requirement of HTTP (Fielding et al., 1999, Section 9)

To prevent non-RESTful implementations, we suggest to add further constraints on the design and implementation level in addition to the constraints defined by Fielding on the architectural level. These constraints result from our experience in implementing several RESTful APIs over the last years. In contrast to Fieldings's Web REST, API developers have to provide a certain degree of security and flexibility and also incorporate every functionality the API user could think of. Naturally, by introducing more constraints we limit the flexibility of developers, which has the consequence that not all RESTful APIs in the meaning of Fielding can be modeled by our approach, compare Figure 1.

In the following we introduce and discuss examples of these constraints. The fixed usage of HTTP methods prevents to model an API, which would operate outside the CRUD principle. In addition, the pagination and query features also limit the user with a fixed amount of returned data per request and the resource filtering depends on the included query parameters. However an experienced developer can extend the generated sources to implement the targeted use case but we believe to have covered most of the use cases.

3.1 Usage of HTTP Verbs

The meaning of HTTP verbs should be fixed and developers should not be free to choose wrong verbs. The four basic operations to create, read, update, and delete resources and mapped to the four HTTP verbs

[3] https://www.jcp.org/en/jsr/detail?id=339

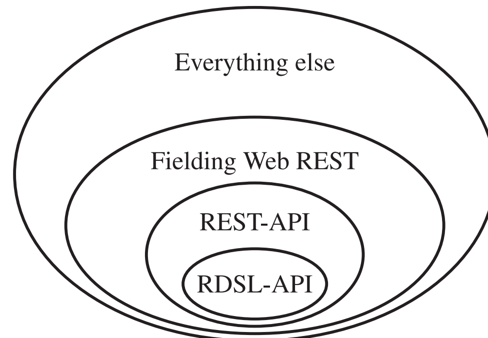

Figure 1: The position of our RDSL-API in relation to current API designs.

POST, GET, PUT, and DELETE. With regard to the HTTP specification, this mapping is unambiguous and not arguable any more. We will enforce a strong coupling of HTTP verbs and business logic.

3.2 Usage of HTTP Status Codes

Every HTTP request is answered by the server with an HTTP status code that informs the client about the success of the last request. The HTTP specification recommends several status codes for typical situations, for example code 200 if a GET request was processed as expected or code 404 if the requested resource is not available. However, in several situations developers are free to select from few codes. For example, in case of a PUT request, the server could reply with code 200 or code 204. The latter code additionally expresses that the response does not contain any content. It should be the decision of the API designer to choose whether the server should reply with code 200 (and with the updated resource as content) or 204 (without any content). The developer should not be able to use code 200 without content.

In case of an error code, it could be advantageous for the client to get more information about the reason for this error. Neither the HTTP specification nor any of the REST principles advise this case. It should be left to the API designer to decide whether a resource representation with further information should be included in the response.

We propose to prohibit wrong usage of HTTP codes while allowing the API designer to select from a set of allowed status codes dependent on the HTTP verb.

3.3 Providing Paging for Collections

Consider an example of a collection of resources that is available at a specific URI path. If the number of

SCIENCE AND TECHNOLOGY PUBLICATIONS

resources becomes too large, fetching the whole collection resource at once becomes too expensive for the client. The server should provide a so-called paging mechanism to allow clients to fetch only a limited number of resources in a single request. We propose to restrict the API designer to choose between two common approaches frequently used:

1. The first approach is to accept two query parameters as part of the URI to let the client define the maximum number of resources (page size) and the first index it wants to receive (offset).
2. The second approach is to use cursors, which is a concept already known from database systems. With cursors, the server and not the client defines the page size. The position of the page to be returned cannot be selected by the client but the server drives the client in fetching batches of resources. As a consequence, the client must always start by fetching the first page.

Both approaches for pagination follow the principles of REST if implemented correctly. This would be the case, if the server provides hyperlinks to the previous or following page as part of the response header (not the response body). On the opposite, the developer can also implement any of these approaches in a wrong way by responding without these information.

However, although we restrict the API designer to use one of these two approaches, we still allow him to define the names of the query parameters for page size, offset, or cursor and to define default values.

3.4 Definition of Resources

The core of a RESTful API are resources and their attributes. Clearly, the API designer must have full flexibility to model resources according to the application domain and use cases. We do not restrict but extend the flexibility compared to programming languages by providing new additional data types. For example, we introduce a data type for location information so that the API designer does not have to model this as an array of two double values for longitude and latitude. We also provide a data types for dates and images that belong to resources.

3.5 Summary

From these examples, it can be seen that the designer of a RESTful interface has many choices when it comes to the implementation using low-level libraries and frameworks. This flexibility might lead developers to choose wrong paths and dead ends, which might finally result in a non-RESTful interface. Our approach is to define further restrictions as part of the meta-model for RESTful APIs while leaving many decisions open to the API designer at the same time. However, these decisions are on a higher level of abstraction and it will be easier for the API designer to decide on them.

4 A META-MODEL FOR THE REST DOMAIN

We created an abstract model of the technical domain of REST based on our understanding of Fielding's constraints (Fielding, 2000) and the recommendations from (Amundsen et al., 2013; Webber et al., 2010; Richardson and Ruby, 2007). The UML based meta-model applies further constraints to the modeling of RESTful APIs. In contrast to other meta-models, our approach focuses on application states and transitions between them. The meta-model serves as abstract syntax of the RDSL developed in Xtext and underpins the code generator written in Xtend.

Hypermedia based APIs can be defined using finite state machines, with pairs of resources and HTTP verbs as states and hyperlinks as transitions between states. As part of a response the server sends hyperlinks to drive clients through the states of the application. Using finite state machines to model application behavior is similar to modeling the design of Web or mobile applications, where techniques such as wireframes or screen-flows are frequently used. An application state can be assigned to a screen and screen transitions are comparable to state transitions, which are the result of user interactions (Thimbleby, 2010).

To concentrate on the state transfer without neglecting other constraints, we divided our meta-model in two parts: the application state and the resource representation.

4.1 Application States

The most important component in our meta-model, Figure 2, is built from resources, HTTP methods, and application states. An application state is a pair of one HTTP method and one resource and represents one valid REST request, which uses a URI to access one resource. An application state is connected to one or more transitions, which link to a target state.

The implementation of the security concept is one of the most time consuming and complex undertakings in the REST field. In our meta-model the API designer has to define the permission feature on the level of application states and has to deal with authorization (e.g. role-based access control or access

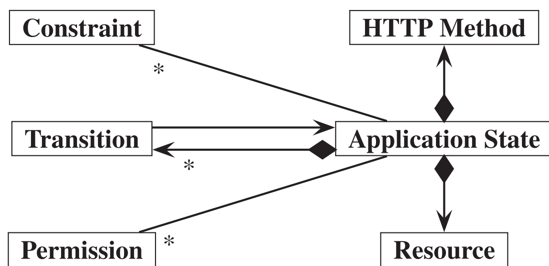

Figure 2: Simplified meta-model component for expressing hypermedia and application state constraints.

control lists) and authentication (e.g. HTTP Basic or OAuth) topics. With the constraint feature it is possible to define rules for accessing application states.

In the model an API designer is able to define global user profiles and override this settings later for all resources separately. The scope of this settings can vary from full access for specific users or roles to restricted data access, for example for guest users without authentication. The user authentication and authorization can be implemented by taking advantage of HTTP security headers or external OAuth APIs.

4.2 Resources

Our resource concept, shown in Figure 3, takes advantage of UML features such as inheritance and associations, resulting in the following resource types: sub-resource, single resource, and resource collection. (Schreier, 2011) identified additional resource types but we were able to shorten the list significantly. For example, we merged Schreier's filter, paging and list resources into the collection resources with query and pagination capabilities. To model the entry point of an API we extended the list of resource types and introduced a new dispatcher resource, which can be still treated as single resource with a list of hyperlinks instead of data. The intention behind the limited amount of resource types is to minimize the complexity of the API, increase its the acceptance for developers, and raise the interoperability to other REST APIs.

Between the three resource types the single and the collection resource differ the most. A single resource contains data attributes, while a collection resource stores multiple resource representations.

The query and pagination features of collection resources allows the API user to filter the response and limit the received amount of data. To specify a query filter, the API designer has to define a list of permitted parameters and the relative path to the query method. The pagination could be implemented either as a cursor based or as a page based pagination as discussed in Sec. 3.3. The actual content of a resource collection could contain only hyperlinks to single resources or embed representations inside and reduce the amount of client calls.

Despite some differences, every resource type has to support a set of common features such as caching, media type, and others with the goal to enforce RESTful design of the produced API. Therefore, we introduce a base element *Resource*, which includes these features and represents a view of a resource. By using independent views, we can implement use cases in which the API has to provide different resource representations depending on the active user or user role.

4.3 Predefined Behavior

We included our understanding of REST into the presented meta-model, which covers all relevant aspects necessary to model a RESTful API. As mentioned previously in Section 3, we limit the API designer in his degree of freedom, for example by predefining URI formats. In the proposed model, the API designer does not need to define URI paths to access resources. The structure of URIs is defined as part of the semantics of the meta-model and later applied by the software generator. Similarly, the API designer cannot influence the functional behavior of HTTP verbs. Depending on the selected method and involved resource component, it can be decided on method parameters, business logic, and proper HTTP responses. For example, in case of method POST the response returns the location of the created resource and there is no room for further interpretation.

5 REST DOMAIN SPECIFIC LANGUAGE

Based on the meta-model presented in the last section, we introduce a domain-specific language (DSL) for REST named RDSL to define an instance of the meta-model. We decided to use a textual language rather than a graphical one in the first step because of its simplicity for novice developers. Graphical representations can be a supplement for understanding the overall concept of the model. We use the Eclipse plugin Xtext (Eysholdt and Behrens, 2010) for defining the grammar of the DSL and the programming language Xtend (Bettini, 2013) to implement the software generator. Xtext also supports the development of DSL-specific editors that provide syntax highlighting as well as checking of syntax and static semantic.

Obviously, modeling of a RESTful application based on finite state machines can also be done using a graphical editor and we are currently working

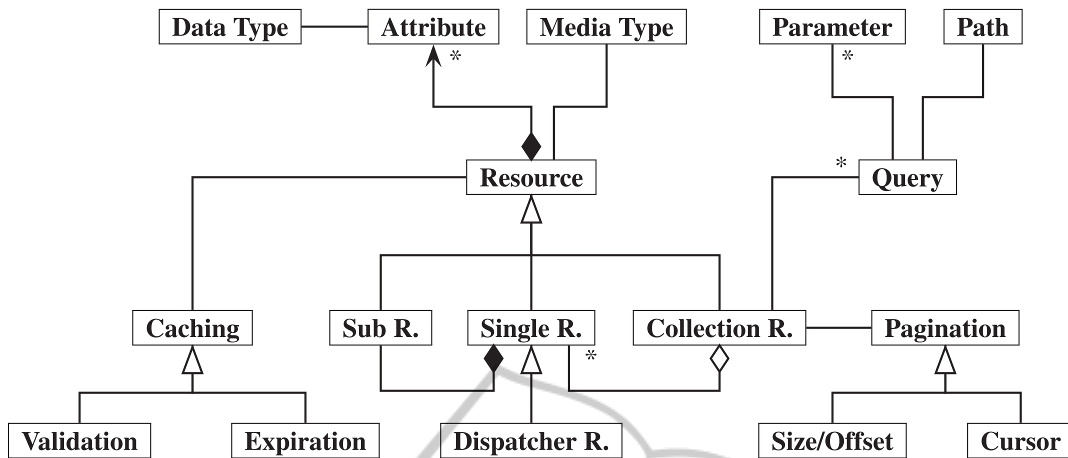

Figure 3: Meta-model with the resource and correlated elements.

on an Eclipse plugin for that based on the Graphical Editor Framework of Eclipse (Rubel et al., 2011).

In the following we introduce the programming language independent RDSL in three steps using a simple example. An API can be completely described in one file of RDSL.

5.1 General Information about the API

Within this `Base` resource, it is defined that all resources will use a unique identifier of data type long. The default media type is set to JSON. At last, different versions of this API can be distinguished by appending a number to the media type.

```
BasePath "api"

AbstractResource Base {
  Long id as key
  MediaType "application/json"
  ApiVersion "1" as MediaType
}
```

Listing 1: Base configuration.

Next, general information about necessary authentication and authorization of users is defined in the `Security` block. User authentication by HTTPBasic is activated, which is linked to resource `User` (which is defined later) using the two attributes `userName` and `password`. For user authorization, a role based access control model (RBAC) is used, which is the only one we have implemented so far. Allowed names for the user roles are defined, as well as the default role for a new user. If a client sends a request without authentication, the anonymous user is assigned to role `other`. By doing so, anonymous access can be permitted for specific states later.

```
Security {
  Authentication by HTTPBasic (User::userName, User::
      password)
  Authorization by RBAC (User::userRole) {
    Roles admin, user, other
    DefaultRole user
    RoleWithoutAuthentication other
  }
}
```

Listing 2: Example for global security definitions.

Finally, as part of the general information, several permissions are assigned to roles. In the first block it is defined that `all`, which is a short-cut for all roles, can access all resources (`*`) with a GET request. In the second block, it is defined that clients with role `admin` can access all resources using all HTTP requests.

```
State (*,GET) {
  Grant Permission to all
}

State (*,*) {
  Grant Permission to admin
}
```

Listing 3: Example for global permissions for states.

5.2 Resources

The next RDSL section handles resources. First, a resource named `User` is defined to extend `Base` with some additional attributes. Caching is defined to be done using a validation model using ETags. In order to let the API provide various resource representations for different user roles, an additional resource view named `DefaultUserView` is defined that contains all attributes of resource `User` but hiding attribute `password` for all users except those who bear the `admin` role. It has to be noted that it is not necessary to define a URI path element for resources,

because the software generator creates them automatically using an approach comparable to (Conway, 1998).

```
Resource User extends Base {
  String userName
  String password
  String userRole
  Caching by ETag
}

ResourceView DefaultUserView < User for user, other {
  Hide password
}
```

Listing 4: Example for defining a resource and a resource view.

The last example in this section shows how to define a collection resource. For this collection, caching is deactivated. A query is defined to accept a URI query parameter `name` that is used to filter by users' name. Sorting is activated to work on the same resource attribute. This information is used later in the generator to create SQL statements. Paging of type `SizeOffset` is activated and the names of the query parameters are defined.

```
Resource AllUsers as collection of User {
  Caching none
  Query SelectByName on path "" {
    Parameter "name" map to "userName" default ""
    Sorting by parameter "order" on "userName"
    optional default "asc"
  }
  Paging by SizeOffset {
    PageSize "size" default 20
    PageOffset "offset" default 0
  }
}
```

Listing 5: Example for a collection resource.

5.3 States and Transitions

Finally, the last RDSL section defines the finite state machine of the application. Each state is defined as pair of a resource name and an HTTP verb. The embedded `Transition` statements specify hyperlinks that are added to the HTTP response header. For example the name `updateUser` defines the relation type and `UpdateExistingUser` is the name of the target state. The software generator resolves the URI to be inserted into the response from the resource used as part of the target state. It is possible to add further `Grant` statements to assign permissions to user roles. A `Constraint` statement defines logical expressions as further guards to access this state.

```
State GetOneUser (User,GET) {
  Transition "updateUser" to UpdateExistingUser
  Transition "deleteThisUser" to DeleteExistingUser
}

State UpdateExistingUser (User,PUT) {
  Grant permission to user
  Constraint $id.userName == $subject.userName
  StatusCode 204
  Transition "getUser" to GetOneUser
}

State DeleteExistingUser (User,DELETE) {
  Grant permission to user
  Constraint $id.userName == $subject.userName
  Transition "getAllUsers" to GetAllUsers
}

State GetAllUsers (AllUsers,GET) {
  Transition "createNewUser" to CreateNewUser
}

State CreateNewUser (AllUsers,POST) {
  Transition "getAllUsers" to GetAllUsers
}
```

Listing 6: Example for states and transitions.

5.4 Software Generation

The software generator gets an RDSL model as input and creates all necessary artifacts for a complete backend that contains the API, the business logic, and the source code of the persistence layer. Right now, the persistence layer for MySQL and RIAK are already available. Fig. 4 gives an overview of all the components that we plan to develop within this project. Currently, we are working on generators for the network layer for mobile applications on iOS and Android.

As platform for the backend we use Java as programming language and Maven as build tool. The Java specification requests (JSR) 339 define a framework to implement RESTful APIs in Java. The reference implementation is Jersey (Gulabani, 2013) which is used in combination with other open source libraries and frameworks for processing XML and JSON. Jersey allows to define own HTTP methods to overcome the shortcomings of the current HTTP 1.1 specification, for example we implemented PATCH (Dusseault and Snell, 2010), LINK and UNLINK methods (Snell, 2013).

The software generator was written in Xtend and is tightly coupled to the Xtext plugin (Bettini, 2013). Fig. 7 partially shows as example of the generated Java source code of method `getOneUser` which corresponds to the state `GetOneUser` from the previous example. The Xtext plugin extracts this information from the text file following the provided DSL and forward them to the Xtend plugin as model, which can be embedded in the code generator templates.

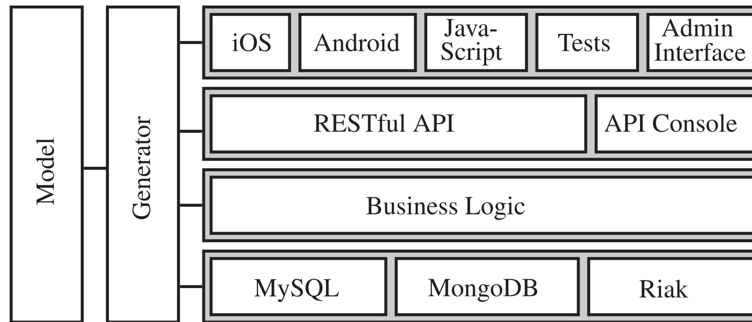

Figure 4: Intended multitier architecture of the generated components from the proposed model.

To generate an access point to the resource `User` following information can be extracted from the model: resource name, resource type, resource identifier, user role, and the expected response. For instance, the path `users` is created automatically from the name of the resource and the `Permission` annotation handles security features for this state as defined in the model. For accessing the database, the data access object pattern is used together with a resource identifier `id` defined in the Listing 1. Hyperlinks defined in the Listing 6 as *Transitions* are embedded in the generated class `UserResponse` and included in the response. This class allows developers to build a RESTful response without worrying about the proper status code or the correct way to include hyperlinks. Developers have to decide to put hyperlinks either inside the response body or the response header but this resource matched class takes care of this task.

```
@Path("/users")
public class UserService extends AbstractService {
 @GET
 @Path("/{id : d+}")
 @Produces({"application/json; version=1"})
 @Permission(userrole = UserRole.Admin)
 public Response getOneUser(@PathParam("id") Long
      id){
  User user = DataAccessObjectFactory.getInstance().
      createUserDAO().load(id);
  ...
  return UserResponse.create(uriInfo).withResult(user).
      build( );
 }
...
}
```

Listing 7: Part of the generated source code for a GET request on resource User.

6 SUMMARY

In this paper we introduced a novel approach for creating RESTful APIs by specifying a model on a higher level of abstraction and generating the source code for the complete implementation of a backend out of it. By this, we overcome two open problems in the development of RESTful APIs.

At first, developers do not need in-depth knowledge about the architectural style of REST as presented in (Fielding, 2000). It is not possible to design an API that contradicts any of them. The code-on-demand, client-server, and layered system constraints can not be broken because of the HTTP usage. The stateless communication is ensured by modeling an encapsulated application state with all the necessary information in it. To reach another application state one transition is required and clients have to send their complete information anew. The cache constraint requires the implicit or explicit labeling of caching, which can be done using the caching structure in our meta-model and the `Caching` keyword in the model. The uniform interface constraint is ensured with the limited amount of methods, their restrictions, and the presence of hypermedia in the response. The API modeler can only use publicly known HTTP methods. The finite state machine based modeling approach protects the hypermedia constraint from Fielding. Therefore, the presented meta-model in Sec. 4 responds to all constraints of REST.

Second, software developers of a RESTful API do not need any knowledge about the specific libraries and frameworks. They benefit from the higher level of abstraction of RDSL and can design the API on the level of application states and resources. However, developers are of course restricted in their flexibility to design an API. We are aware of the fact that it is not possible to describe all RESTful APIs with RDSL. For example, we only provide two ways to implement a paging mechanism for resource collections, although there exist many more that are also RESTful. We will learn more about the severity of this restriction in next projects and extend the meta-model if necessary.

6.1 Reflections on the Approach

The project is still under development and bound to the generation of straightforward REST APIs with focus on data and ability to manipulate it. The API modeler can add attributes to resources, which will also be included in the JSON representation. Furthermore, the generators produce only Java source code right now. In a production environment, support of other languages and platforms might be necessary. Implementing new generators and managing them requires moderate additional work.

Without a proper client the usage of REST and especially hypermedia provided in the response is not guaranteed. The server response is defined by our understanding of REST and it includes hyperlinks to related resources or actions. Clients could skip this additional information and ignore the hypermedia principle completely. One of the biggest issues is the completeness of RDSL, which can not be assured. With new features (e.g. client source code, testing, developer console) there is a chance of a rule-breaking modification, which could result in rewriting the DSL, the generators, and everything depending on it. This was confirmed by two students, who tried to add the support for NoSQL database and API testing.

6.2 Outlook

Ongoing work focuses on the development of a graphical editor as Eclipse plugin in order to define at least the finite state machine of the application. Future work will be done mainly in the area of generating source code for mobile applications. As already shown in Fig. 4 we target the mobile platforms iOS and Android. The goal is to generate libraries to simplify the communication to the RESTful API by providing business classes and methods to wrap all HTTP requests. Another goal is to generate a Web-based administration interface for inspecting and modifying the current status of the backend.

REFERENCES

Amundsen, M., Richardson, L., and Ruby, S. (2013). *RESTful Web APIs*. O'Reilly Media, Inc., Sebastopol, CA, USA.

Bettini, L. (2013). *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, Birmingham, United Kingdom, first edition.

Conway, D. (1998). An algorithmic approach to english pluralization. In *Proceedings of the Second Annual Perl Conference*, San Jose, USA.

Costa, B., Pires, P., Delicato, F., and Merson, P. (2014). Evaluating a representational state transfer (rest) architecture: What is the impact of rest in my architecture? In *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*, pages 105–114, Sydney, NSW.

Coulouris, Dollimore, J., and Kindberg, T. (2005). *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Dallas, A. (2014). *RESTful Web Services with Dropwizard*. Packt Publishing, Birmingham, United Kingdom, first edition.

Dusseault, L. and Snell, J. (2010). PATCH Method for HTTP. RFC 5789, Internet Engineering Task Force.

Eysholdt, M. and Behrens, H. (2010). Xtext: Implement your language faster than the quick and dirty way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 307–309, New York, NY, USA. ACM.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California.

Fielding, R. T. (2008). REST APIs must be hypertext-driven. URI: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven. Last visited 12.12.2014.

Fielding, R. T., Gettys, J., Mogul, J. C., Frystyk, H. N., , Masinter, L., Leach, P. J., and Berners-Lee, T. (1999). Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force.

Fielding, R. T. and Taylor, R. N. (2002). Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150.

Gulabani, S. (2013). *Developing RESTful web services with Jersey 2.0*. Packt Publishing, Birmingham, United Kingdom.

Klein, U. and Namjoshi, K. S. (2011). Formalization and automated verification of restful behavior. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 541–556, Berlin, Heidelberg. Springer-Verlag.

Laitkorpi, M., Selonen, P., and Systa, T. (2009). Towards a model-driven process for designing restful web services. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 173–180.

Lanthaler, M. and Gutl, C. (2010). Towards a restful service ecosystem. In *Digital Ecosystems and Technologies (DEST), 2010 4th IEEE International Conference on*, pages 209–214, Dubai. IEEE.

Papazoglou, M. P. (2008). *Web Services - Principles and Technology*. Pearson Education Limted, Harlow, United Kingdom.

Pautasso, C., Zimmermann, O., and Leymann, F. (2008). Restful web services vs. "big"' web services: Making the right architectural decision. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 805–814, New York, NY, USA. ACM.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Pérez, S., Durao, F., Meliá, S., Dolog, P., and Díaz, O. (2011). Restful, resource-oriented architectures: A model-driven approach. In *Proceedings of the 2010 International Conference on Web Information Systems Engineering*, WISS'10, pages 282–294, Berlin, Heidelberg. Springer-Verlag.

Porres, I. and Rauf, I. (2011). Modeling behavioral restful web service interfaces in uml. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1598–1605, New York, NY, USA. ACM.

Richardson, L. and Ruby, S. (2007). *Restful Web Services*. O'Reilly, Sebastopol, CA, USA.

Rubel, D., Wren, J., and Clayberg, E. (2011). *The Eclipse Graphical Editing Framework (GEF)*. Eclipse (Addison-Wesley). Addison-Wesley.

Schreier, S. (2011). Modeling restful applications. In *Proceedings of the Second International Workshop on RESTful Design*, WS-REST '11, pages 15–21, New York, NY, USA. ACM.

Snell, J. M. (2013). HTTP Link and Unlink Methods draft-snell-link-method-10. Technical report, Internet Engineering Task Force. Expires: 02.02.2015, Last visited 12.12.2014.

Stahl, T., Voelter, M., and Czarnecki, K. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, Inc.

Tavares, N. A. C. and Vale, S. (2013). A model driven approach for the development of semantic restful web services. In *Proceedings of International Conference on Information Integration and Web-based Applications & Services*, IIWAS '13, pages 290:290–290:299, New York, NY, USA. ACM.

Thimbleby, H. (2010). *Press On: Principles of Interaction Programming*. The MIT Press.

Webber, J., Parastatidis, S., and Robinson, I. (2010). *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, Inc.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS