Hypermedia: The Key to RESTful Web Applications

Patrick B. M. Müller, Tobias Fertig, Henry Vu and Peter Braun
Faculty of Computer Science and Business Information Systems, University of Applied Sciences Würzburg-Schweinfurt, Sanderheinrichsleitenweg 20, 97074 Würzburg, Germany

Keywords: Model-driven, MDSD, Metamodel, Web Engineering, Web Components, Polymer, Graphical User Interfaces, REST, RESTful Web Clients, Hypermedia.

Abstract: Implementing the hypermedia constraint for RESTful Systems is a challenging task for most developers. This is due to the lack of information about how to implement hypermedia on the client-side correctly. Therefore, new tools are required to support developers with the implementation of the hypermedia constraint. We propose a model-driven approach that allows developers to define a RESTful System as a finite-state machine: states represent resources and transitions represent hypermedia links. We present a metamodel that our generator can process to generate Polymer Web Applications. These web applications fulfill all REST constraints defined by Fielding. Therefore, developers do not have to implement the hypermedia constraint themselves. Our approach simplifies the development of RESTful Web Applications and reduces the development efforts. Moreover, we describe how RESTful Web Applications can be derived from finite-state machines.

1 INTRODUCTION

Over the past few decades, the World-Wide Web has become a global information system. This was enabled by the scalable and reliable architecture of the Web. It was designed for the rapid growth of users and applications without knowing the term hypermedia. The precursor of hypermedia was then called hypertext, which is why the Hypertext Transfer Protocol (HTTP) was specified.

In 2000, Fielding derived Representational State Transfer (REST) as the architectural style for distributed hypermedia systems in his thesis (Fielding, 2000). This architectural style meets the requirements of the modern Web and can be combined well with HTTP. Fielding defined six constraints that determine how a web application should behave to ensure certain quality criteria. The quality criteria include scalability and performance requirements. In his thesis, he defines five mandatory and one optional constraint. However, he just explains those constraints and gives no guidance or recommendation on how to implement them. Therefore, false interpretations were made, which is why developers misunderstand the REST constraints. For this reason many web applications do not meet the quality requirements anymore.

The Hypermedia As The Engine Of Application State (HATEOAS) constraint was most misunderstood and discussed. HATEOAS describes how the client and the server should communicate via hypermedia, which is why the constraint is often referred to as the hypermedia constraint. Many developers believe the constraint is optional, but this is denied by Fielding in his blog (Fielding, 2008). Fielding himself defined the hypermedia constraint as mandatory, while in contrast the Richardson Maturity Model defines hypermedia as the highest degree of RESTful Applications (Webber et al., 2010). Therefore, Richardson implies the hypermedia constraint to be optional, which was not ever the intention of Fielding.

The implementation of all REST constraints presents many developers with a great challenge. So we are looking for an approach to simplify the development of RESTful Web Applications. Formal models can support developers by providing an abstract view of software systems. Model-driven Software Development (MDSD) also uses formal models to generate code (Völter et al., 2013). A similar approach based on formal models are the Low Code Development Platforms (Richardson et al., 2014).

Schreibmann et al. proposed a model-driven approach to generate RESTful APIs using metamodels (Schreibmann and Braun, 2015). These metamodels describe the REST domain using finite-state machines (ε-NFAs) after Zuzak et al. (Zuzak et al., 2011). These ε-NFAs are easy to understand and can be used to describe RESTful Systems on a higher level of abstraction. However, developers still have

Müller, P., Fertig, T., Vu, H. and Braun, P.
Hypermedia: The Key to RESTful Web Applications.
DOI: 10.5220/0006927103780387
In *Proceedings of the 14th International Conference on Web Information Systems and Technologies (WEBIST 2018)*, pages 378-387
ISBN: 978-989-758-324-7

to implement hypermedia-driven frontends by themselves. Since there is not much information available about how to build RESTful APIs correctly, there is even less information about hypermedia-driven clients. Developers are facing the challenge of implementing the complex hypermedia constraint. Therefore, we propose a MDSD approach to generate RESTful Web Applications that fulfill all REST constraints. The generator is an extension to our model-driven approach proposed by Schreibmann et al.

Within this work we are focusing on the following research questions:

Q1) How can RESTful Web Applications be derived from finite-state machines?

Q2) How can a model-driven approach support developers building RESTful Web Applications?

Q3) Can the hypermedia constraint be fulfilled transparently without manual efforts of individual developers?

To answer our research questions we will summarize related work in Section 2. We will show that models can ease the development of software systems. Afterwards, we will give a short summary of the definition of finite-state machines for RESTful Systems. Moreover, we will give an overview of existing approaches for generating web applications. In Section 3 we summarize our project GeMARA and give a short introduction into how our metamodel works. In Section 4 we summarize the different challenges we had to face during our work. Afterwards, in Section 5 we propose our model-driven approach. We used Polymer as a library for web components (Evans, 2015) and solved the previously mentioned challenges. Finally, in Section 6 we summarize our results, give answers to our research questions, and show some limitations of our generators. We end with a summary of future work and how we will continue to use our findings.

2 RELATED WORK

Models are a common practice in software engineering. Already in 1992 Jacobson et al. proposed modeling as tool for object-oriented programming (Jacobson, 1992). Moreover, in 1995 relational models were introduced for databases in the third manifesto (Darwen and Date, 1995). Since then and maybe even earlier engineers routinely create models when analyzing and designing complex systems in order to abstract those systems and their environment.

The adjective *driven* in MDSD emphasizes that this paradigm assigns models a central and active role. In other words models are as important as source code. MDSD is a discipline in software engineering to generate runnable source code from metamodels that allow domain developers to more accurately describe the domain's problem space by abstracting from the underlying programming language (Völter et al., 2013).

Therefore, we required a formal model to describe REST. Zuzak et al. presented a model of RESTful Systems based on a finite-state machine (ε-NFA) formalism, which should create a better understanding of the architectural style REST (Zuzak et al., 2011). They described three main parts of ε-NFA operations: the *Input Symbol Generator*, the *Transition Function* and finally the *Current State*. Their mapping of REST to an ε-NFA can easily be transformed into a metamodel. However, they mainly focused on GET requests in their work. So we had to extent the model for the other HTTP verbs as well. Afterwards, we had a solid metamodel to abstract from the complex architectural style REST.

There is a lack of information about how to implement hypermedia on the client-side. Besides Mike Amundsen's book there are no recommendations for building effective hypermedia-based client applications (Amundsen, 2017). Amundsen's approach should lead to a reduction of custom client code, as his approach works in general for every RESTful System. He recommends that a server emits templates that are processed by a client. The client uses the templates to render the Graphical User Interface (GUI) of the web application. If actions for updating or deletion are included in the template, the client will render the associated buttons. In this case, the backend developer decides what to display in the frontend. Frontend developers cannot influence the appearance of the web interface. Because his approach is not model-driven, the backend developer is responsible for implementing the REST constraints correctly. Thus, even with Amundsen's approach, a faulty implementation of the REST constraints is still possible. Moreover, Amundsen did not use any formal models we could use for our model-driven approach. Therefore, we had to implement a different approach to enable hypermedia on the client-side.

The idea to generate web frontends is not a new one. In 2003 Shimomura et al. presented their approach to generate web applications based on images (Shimomura et al., 2003). The motivation was to ease the development process and help developers to built web applications faster. In 2007 Jakob et al. released a graphical modeling tool to define models and generators for data management web applications (Jakob et al., 2007). The past shows that generators for web applications worked for different domains. Even no-

wadays more and more approaches for frontend generation are proposed. Beltramelli proposed an approach to generate GUIs. His approach is based on artificial intelligence, which analyzes a screenshot of the desired GUI and uses it to generate the code for the web application (Beltramelli, 2017). The disadvantage with this method is that a GUI must already exist - at least in form of a screenshot. Because Beltramelli works independently of the REST domain, and we cannot provide screenshots for every RESTful Web Application, we still need a different approach.

Another approach to generate frontends was proposed in 2018 by Jaber et al. They developed a metamodel that describes GUIs for Android Applications at a higher level of abstraction (Jaber et al., 2018). The developers use their metamodel to determine which control widgets to display on each view of the Android Application. A generator processes this metamodel to generate the code of the Android Application. Unfortunately, this metamodel is not suitable for our purposes because of the technical reference to the Android domain. Furthermore, Android's programming model is very different from conventional web development techniques. Additionally, the metamodel lacks important REST specifications because states, resources, or attributes are not considered. For these reasons, we decided to design our own metamodel for generating RESTful Web Clients.

There are many proven techniques and practices in the literature that support the successful implementation of a model-driven solution. Voelter et al. recommended an approach that is divided into three steps (Völter et al., 2013): First, building a reference implementation. Second, separate domain-specific and generic code. Third, build the generator for the domain-specific code. We used this approach to build our own metamodel and our own generator for RESTful Web Applications. We chose MDSD because this approach not only reduces the burden on the development team but can also improve the quality, efficiency, and predictability of large-scale software development due to its automation potential (Kelly and Tolvanen, 2008).

3 GeMARA

The goal of our research project is to develop Generators for distributed Mobile Applications based on RESTful Architecture (GeMARA) using a model-driven approach. Accordingly, applications are no longer developed individually, but are described by an abstract model from which source code and other artifacts are generated.

We proposed our approach in 2015 (Schreibmann and Braun, 2015). Back then we were able to generate RESTful APIs including the persistence layer. Our generators were written with Xtext (Efftinge, 2014b) and Xtend (Efftinge, 2014a). For the architecture we used the recommendations of Bettini (Bettini, 2013). However, as our project matured, the maintenance of our generators got a time consuming task. Therefore, we cut loose from Xtext and Xtend and developed our own internal Domain-specific Language (DSL) in Java (Fowler, 2010). The latest DSL describes RESTful Systems as ε-NFAs according to (Zuzak et al., 2011):

States are defined by the HTTP Verb and the resource representation. Transitions represent the hypermedia links. Subresources can also be defined within our metamodel. With those subresources we can describe relations between resources, for example between users and their addresses.

We can generate every RESTful System described as ε-NFA. By now it is possible to generate web clients. Moreover, we are generating test cases for the RESTful API (Fertig and Braun, 2015) and hypermedia tests (Vu et al., 2017). The generated source code can be deployed out of the box. However, sometimes manual adjustments are necessary. Those are also supported by our approach using dependency injection.

In order to generate RESTful APIs, further restrictions of the REST domain are necessary. The six constraints defined by Fielding (Fielding, 2000) are not enough to apply a model-driven approach. Therefore, we had to define a subset of the REST domain. We can guarantee that the generated RESTful Systems fulfill the REST constraints. However, it is possible that there are RESTful Systems we cannot generate in the same way they were written manually before. These restrictions of constraints mainly affect the types of states. We believe that a RESTful API must support at least the following five state types: *GetCollectionResource, GetSingleResource, PostResource, PutResource,* and *DeleteResource*. Figure 1 shows an example of a lecturer domain modeled as a ε-NFA.

These states can be connected according to a fixed pattern via transitions which gives a generic structure of a RESTful API. We also divide the ε-NFA states into a primary and secondary level. States of the primary level offer functionalities users can use to retrieve or manipulate main resources. Secondary level states can be used to link resources to other resources. The secondary level can be reached after a resource has been retrieved at the primary level - via the *GetSingleResource* State. Table 1 shows the functionali-

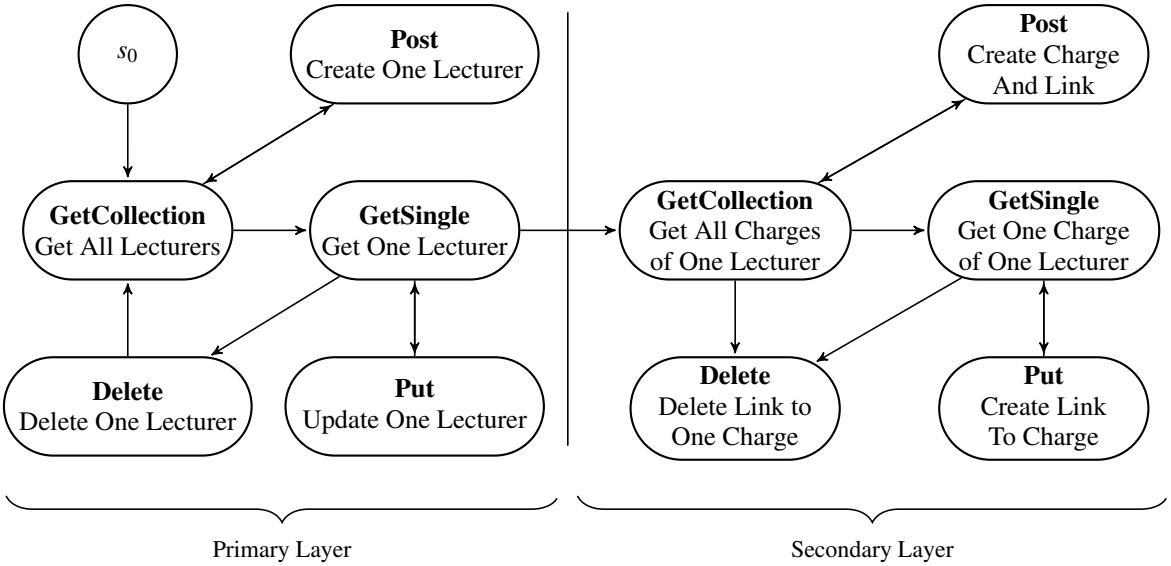

Figure 1: The ε-NFA for the lecturer domain. On the left, there is the 'Primary Layer' containing all states for the manipulation of lecturer resources. On the right, there is the 'Secondary Layer' containing all states for the manipulation of links between lecturer resources and their charges. The arrows represent possible transitions between states. A client can always return to the s0 state: transitions were removed for sake of readability.

ties offered by the state types. Depending on which level a state is located, the functionality changes.

The states are connected by transitions which show the client the next possible states. Transitions consist of a *URL*, a *RelationType* and a *MediaType*. The URL indicates where the state of a resource is located. Based on the relation type, the client can decide which HTTP verb to use for the request. The media type tells the client which representation of the resource to use. We can set a constraint for each transition to allow filtering links based on the users' roles. The associated hypermedia links are only delivered if the constraint is fulfilled. We always define the maximum ε-NFA, but we can always omit individual states if the permissions are not sufficient. If some of the state types are not wanted for certain resources, the domain expert simply cannot model them and adapt them to their wishes.

Based on these ε-NFAs, GUIs can be generated. For this, we are taking advantage of the new standard for web components (Cooney, 2014). Frontend developers are no longer responsible of implementing REST by themselves. They no longer have to focus on the correct behavior according to Fielding, they can now focus on describing the web application as an abstract model.

4 CHALLENGES

Since our research question is *how RESTful Web Applications can be derived from finite-state machines*, we have to face different challenges. In order to solve these challenges we try to take advantage of the benefits of the Web Components Standard (Cooney, 2014). Therefore, we analyze resources which are the key abstraction of information in RESTful Systems (Fielding, 2000).

Resources can have different representations that consist of simple properties and links to other representations (Richardson and Ruby, 2008). How can a resource with its properties be graphically displayed in a web interface? Established design patterns such as the *Master Detail View* (Jovanovic, 2011) do not provide a clear answer to this question. The master list view can be represented by the *GetCollectionResource State Type*. The detail view could then be represented by a *GetSingleResource State Type*. The Design Pattern, however, lacks the concept of input masks that allow users to create or update a resource. Furthermore, it remains unclear how the design pattern could be used to easily link resources. So we need to develop our own approach on how to map the states of a RESTful API within a web application.

The exchange of resources between client and server causes state transitions within the ε-NFA (Zuzak et al., 2011). The client manages the current state of the communication in its application state. During a state transition, the server signals the next possible

Table 1: An overview of functionalities offered by different state types. The column 'Primary' explains the functionalities on the primary layer, whereas the column 'Secondary' explains the functionalities on the secondary layer.

State Types	Primary	Secondary
GetCollectionResource	Representation retrieval of a collection of resources.	Retrieval of a collection of all linked or unlinked subresources depending on the chosen mode
GetSingleResource	Representation retrieval of a single resource	Retrieval of a single linked resource
PostResource	Creation of a resource	Creation of a new resource including a link to an existing resource
PutResource	Update of a resource	Creation of a link between two existing resources
DeleteResource	Deletion of a resource	Deletion of a link between two resources

transitions from which the client can select one. In order to implement the hypermedia principle correctly in the client, the web application must detect a transition within the hypermedia links of the response. Moreover, the presence of such a link must be presented to the user. This can be done by showing or hiding buttons (Amundsen, 2017). For example, if the currently logged-in user is allowed to create a resource, the server will emit a corresponding hypermedia link in the response header. The hypermedia link leads to the state for creating a resource. This will allow the client to display a button for resource creation in the GUI. The transitions determine the navigation concept of the web application.

To generate executable applications automatically, restrictions have to be made in order to give the generator a framework. A model-driven approach is more effective if several related systems are to be developed that can be grouped together under one software system family (Völter et al., 2013).

A model-driven approach requires encapsulation of the generic and domain-specific code sections of an application. In object-oriented programming languages, class concepts are available that enable encapsulation. This feature is easy to use to encapsulate related code. In web development such constructs are not available. Conventional JavaScript libraries provide only limited support to outsource the code of a web application into individual files (Haverbeke, 2011). Although smaller JavaScript, HTML, and CSS files can be created, this type of encapsulation is not desirable for a model-driven approach. This concept does not define what a reusable component might look like, so it is not suitable for MDSD.

Recent web technologies are introducing component-based approaches according to the Web Components Standard (Cooney, 2014). This allows code sections to be easily separated from application-specific sections. However, we still do not know how a model-driven architecture for web components can be designed.

5 APPROACH

In order to generate RESTful Web Applications, we need to commit to a fixed architecture. Our approach is based on the generic structure of RESTful APIs introduced in Section 3. We assume that every state of the ε-NFA is represented by a separate view in the web application (Zuzak et al., 2011). In the following, we clarify which views are to be created for the individual states from Figure 1 on the basis of the *Lecturer Domain*. We explain how RESTful Web Applications should behave and propose our metamodel for describing their views. Finally, we give some insights into the generator that creates web applications based on Polymer web components.

5.1 Analysis of States

The *s0* state in Figure 1 represents the dispatcher state that embodies the entry point of the generated API. The URL of the dispatcher state is the only one known to the client. The dispatcher state uses transitions to point to the next possible *PrimaryGetCollection States* and can be represented in the web application by a *navigation drawer menu* that users use to navigate between the *PrimaryGetCollection States*.

A *PrimaryGetCollection State* can be represented by a `ListView` that displays a collection of resources. Each resource is represented by a `CardView`. In the `ListView`, a button is displayed so that users can navigate to an input mask to create new resources.

As already mentioned, a *PrimaryPost State* can be mapped to an `InputView` within the web application. It consists of input widgets that allow users to more easily set the attribute values of the resource to be created. Depending on the attribute type, different widgets must be provided. The attribute type has to be defined by using the metamodel.

A *PrimaryGetSingle State* is represented by a `DetailView` that is displayed after the user clicked on a `CardView` within the `ListView`. In the

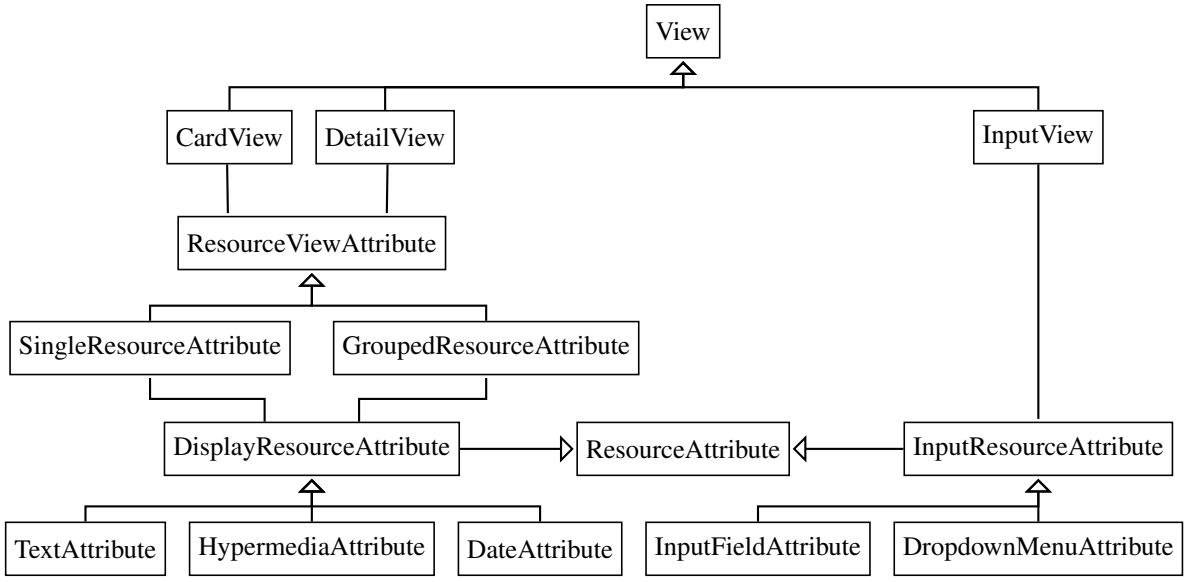

Figure 2: Simplified version of our metamodel as UML class diagram.

`DetailView`, the attribute values of the resource are also displayed differently depending on the attribute type. A date must be displayed in the correct format, while a hypermedia link must be displayed as a clickable link. Again, the attribute types must be modeled by using the metamodel.

The corresponding web interface of *PrimaryPut States* is similar to the `InputView` for creating a new resource. However, all widgets are preallocated with the resource's current attribute values. The user can then just change the attributes that need to be updated.

We do not provide additional views for *PrimaryDelete States* in the web application, since a generic dialog is sufficient here. This dialog asks for the users confirmation to delete the resource.

A *SecondaryGetCollection State* is represented by a `ListView` that displays all linked subresources of a resource. This view can only be reached after the main resource was requested from the *PrimaryGetSingle State*. Table 1 shows that the states located at the secondary level provide functionality for linking and unlinking resources. To simplify the linking or unlinking of resources, the `ListView` has two different modes: In the *OnlyLinkedSubresources* mode, the `ListView` displays all linked subresources. The second mode is the *AllPossibleSubresources* mode, which displays all resources that are already linked and the ones that can be linked. Each `CardView` displays a check box so that subresources can be easily linked or unlinked from the main resource.

SecondaryPut States are embodied by `InputViews` that are used to update the subresource and establish a link to the main resource.

SecondaryPost States are also represented by `InputViews`. These enable the creation of a subresource that will be automatically linked to the main resource.

Clicking on a `CardView` within the `ListView` opens the `DetailView` of the resource. This view is identical to the view of *PrimaryGetSingle States*.

A generic delete dialog can also be used to represent *SecondaryDelete States*. This state offers functionalities to remove a link to the main resource. The subresource itself will not be deleted.

The analysis of the different states shows that four different views are required: `CardViews`, *DetailViews*, `InputViews`, and `ListViews`, that are represented by a list of `CardViews`. Now we want to present our understanding of how RESTful Web Applications should behave. The `ListViews` that represent our *PrimaryGetCollection States* can display a button that can be used to navigate to a POST `InputView`. Due to the hypermedia constraint this button may only be displayed under the following conditions:

- The ε-NFA includes a transition to a corresponding *PrimaryPost State*, which offers the POST functionality.
- The client receives a link in the response header that contains the POST URL.

The web application is responsible to analyze the response link headers to determine if the server allows the transition to the *PrimaryPost State*. If the server does not allow the creation of a resource, the POST button must not be displayed to the user. Therefore, the implementation of the hypermedia constraint re-

quires a fixed structure of the ε-NFA. The web application uses the provided link headers to decide if the navigation buttons must be presented to the user. This behavior is what we consider as a RESTful Web Application.

Now that we have determined how a ε-NFA affects a RESTful Web Application, we will present our metamodel.

5.2 Metamodel

By using the metamodel domain experts can describe each view of a web application at a higher abstraction level. This metamodel can be processed by a software generator to create functional web applications. The key abstraction of information in REST is a resource that has attributes in the form of key-value pairs (Fielding, 2000). Because values of attributes are shown to the user, domain experts must use the metamodel to define how these values should be represented.

We examined GUIs from a non-technical perspective to keep the metamodel generic. Because of this, other generators can use this metamodel to create client applications based on other frontend technologies.

The easiest way to represent a resource in a view is to arrange its attributes vertically one by one. For this reason, a vertical layout is used in our views. In `CardViews` and `DetailViews` it may be necessary to display attributes horizontally next to each other. This is the case if the full name of a lecturer consists of the academic title, first name, and surname. In `InputViews` the possibility of a horizontal arrangement should be denied, since otherwise rendering problems may occur on mobile devices. These often have smaller screens and are therefore not suitable for horizontal arrangements of input widgets.

The order in which the attributes are arranged can be determined by the domain expert. In addition, they can use the metamodel to define which font size and color should be applied to the attribute values. Figure 2 shows a simplified version of our metamodel.

The metamodel defines how the attributes of a resource should be graphically displayed in the interface. For this reason, we start by explaining the class `ResourceAttribute`. This represents an attribute of a resource in the metamodel.

In the metamodel, we distinguish between attributes that can be read or written. `DisplayResourceAttributes` represent the read-only attributes, while `InputResourceAttributes` represent the modifiable attributes. These two classes serve as superclasses for specific attribute types, which can be seen in the lowest level of the diagram in Figure 2.

Depending on the attribute type, the attribute value must be processed differently in the client application. We provide some examples that show why different attribute types are necessary:

- A first name as text can be displayed directly as a string. In the `DetailView` one would use a `TextAttribute`. In the `InputView` one would use an `InputFieldAttribute`.
- Hypermedia links must present a clickable link in the GUI.
- Timestamps have to be converted into a human-readable format.

These are just a few obvious examples of possible attribute types. Our metamodel can easily be extended by inheritance for additional attribute types.

A `DisplayResourceAttribute` can be wrapped in a `SingleResourceAttribute`, informing the generator that the attribute should occupy an entire row within the vertical layout. If several attributes of a resource are to be displayed horizontally next to each other, they must be grouped using a `GroupedResourceAttribute`.

With the metamodel `CardViews`, `DetailViews`, and `InputViews` can be described. The `CardViews` are required within `ListViews`. Delete views do not have to be generated separately because a generic delete dialog is sufficient. The `CardViews` and `DetailViews` have a collection of `ResourceAttributes`, which represents the content of each view. Since no input widgets can be grouped horizontally in `InputViews`, `InputResourceAttributes` cannot be grouped by `GroupedResourceAttributes`.

The generator now has all the information needed to generate the views required by a RESTful Web Application based on this metamodel. Each state of the ε-NFA is mapped to a specific view description with the exception of the delete states. For example, a PUT state receives an instance of an `InputView`. The collection `InputResourceAttribute` describes the content of the input mask that has to be generated.

5.3 Generators

For the implementation of a model-driven approach, Voelter et al. proposed to start developing a reference implementation (Völter et al., 2013). The reference implementation, in our case, is a manually developed web application that communicates with a RESTful API. The API provides functionalities for managing lecturers that work at a faculty. For each state of the

```
<template>
  <abstract-card-view
    url="[[url]]"
    update_rel_type="updateLecturer"
    delete_rel_type="deleteLecturer">
  </abstract-card-view>
</template>
<script>
  Polymer( {
    is: "lecturer-card-view",
    properties: {
      url : { type: String, value: "" }
    } } );
</script>
```

Listing 1: Excerpt from the 'lecturer-card-view' component. The code highlighted in orange is domain-specific information that is inserted from the metamodel into the generator template (black).

ε-NFA, a separate view was implemented as described in Section 5.1. The web application has implemented all the required features needed in the generated web applications.

The next step is to analyze the code of the reference implementation more closely to identify the generic and domain-specific code sections. The generic code is the same in all generated web applications. To reuse this code, the generic code sections must be separated into different components. The domain-specific code is different in each project and therefore must be generated by the generator using the metamodel. We decided to use the Polymer library to separate these two code sections. Polymer is based on the concept of web components, whereby the necessary separation of the code sections can be realized.

In the following, we explain how the two code sections can be separated using the example of a `lecturer-card-view`. The `lecturer-card-view` represents a resource on a `CardView` within a `ListView`. `CardViews` play an important role in our navigation concept, as they provide buttons that allow the user to update or delete the resource. The `lecturer-card-view` component requires the relation type for PUT and DELETE requests to implement the hypermedia constraint. Only then the client can determine the URLs for updating and deleting the resource. If the client finds the corresponding URLs in the response link headers using this relation types, the buttons have to be displayed.

We now form two components: the generic `abstract-card-view` and the domain-specific `lecturer-card-view`. Our `abstract-card-view` is the same in all web applications. Therefore, it can be reused in every project. It contains the logic needed in every `CardView`. These include the following functionalities:

- The network communication to request the resource from the server.
- The analysis of the response link headers to display the corresponding buttons.
- The navigation logic to the `UpdateView` or `DeleteView`.

The domain-specific `lecturer-card-view` contains all the information about how the attributes of the resource should be displayed. This component must be generated because the information about the view content is described in the metamodel.

Due to the necessary code separation, the generic and domain-specific sections of the `CardView` were separated into two different components. In order to establish the communication between these two components they must be properly nested. The nesting is enabled by using the content tag offered by the Polymer library. We illustrate this concept within Figure 3.

According to this approach, the domain-specific `lecturer-card-view` encloses the `abstract-card-view` component. The inner component requires domain-specific information that must be set by the outer components via its properties. The generated HTML code will be inserted into the generic component by using the content tag. Listing 1 shows that the generic component now has all the information needed to request resources from the server and present the resource to the user.

These concepts also apply to the other required components, such as `ListViews`, `DetailViews`, and `InputViews`. The generator iterates over the states of the ε-NFA and generates the required components for each state. The generated components are arranged by the generator to create functional web applications.

6 DISCUSSION

At the beginning we summarized related work and explained that modeling has a central role in software development. Moreover, we showed that model-driven approaches for developing frontends already worked in the past. Afterwards, we introduced our project and explained how to define RESTful Systems as ε-NFAs. After having clarified all challenges, we presented our approach and outlined results.

To answer our first research question Q1, we take a look at the results of Section 5.1. In order to implement RESTful Web Applications we need four different views within our approach: `CardViews`, *DetailViews*, `InputViews`, and `ListViews`, that are represented by a list of `CardViews`. Moreover, the frontend is responsible for analyzing the response link headers

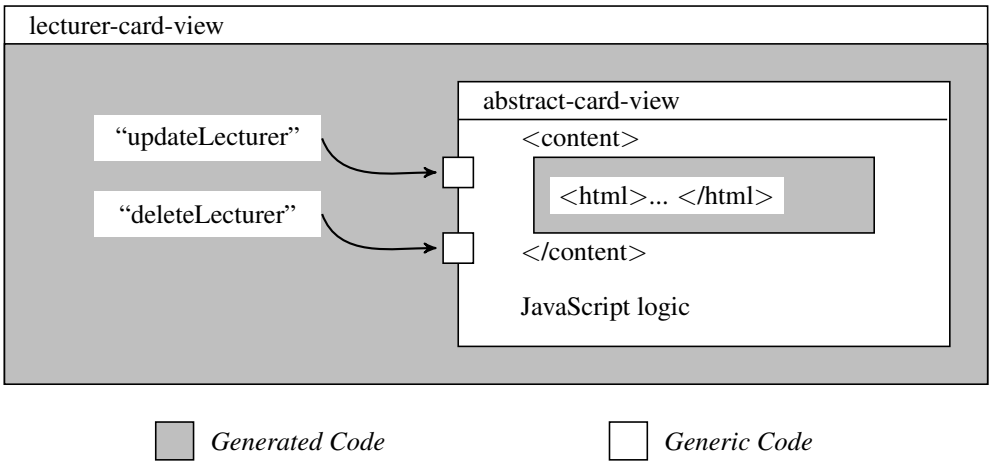

Figure 3: This figure shows how the generator fills the missing information into our code templates. The ‘abstract-card-view’ contains all generic logic. Our generator generates the ‘lecturer-card-view’ and inserts the properties from the metamodel into the generator template shown in Listing 1. The domain-specific information is transferred to the generic component by setting its properties. The generic component can then access this information which will be required for the network communication.

to determine which buttons must be displayed. This behavior fulfills all six constraints of REST. Furthermore, the four views were sufficiant to describe all requirements within our lecturer domain in our metamodel.

The advantage of this abstraction into four different views is that new developers can be onboarded fast. Nevertheless, it is not always trivial to divide domain requirements into only four views. Supporting more views could be helpful and ease the development even further. Therefore, we need to investigate if additional views are supported by the REST constraints.

The research questions Q2 and Q3 both focus on supporting developers via a model-driven approach. Since ε-NFAs are easier to understand and easier to define than implementing RESTful Systems from scratch, our metamodel can reduce the development efforts. Developers can easily define all hypermedia aspects via the state transitions in our ε-NFA. They do no longer require any knowledge on how to implement the hypermedia constraint correctly. They only need to know the hypermedia principle itself and how to describe RESTful Systems as ε-NFA. We have shown that viable web applications can be realized with a model-driven approach. Especially within the domain REST, the model-driven approach is suitable because REST is very restrictive and repetitive. This results in a high degree of reuse (Kelly and Tolvanen, 2008). Web components are also very well suited because the required encapsulation of the code is enabled.

A disadvantage of our current metamodel is the fixed structure of our ε-NFAs. Nevertheless, we need a fixed structure in our model-driven approach in order to generate web applications. Any change to the fixed structure would require all generators to be rewritten. The drawback is that our web application cannot support every possible RESTful API which is in conflict with Amundsen’s vision (Amundsen, 2017).

Another drawback is that the metamodel is currently very technical. Developers still have to understand how to describe REST as ε-NFA, and our metamodel could benefit from further abstraction levels. We already have initial approaches to create a higher abstraction of our metamodel on the server-side (Ulsamer et al., 2018). It would be interesting to explore the possibility of abstraction levels for the description of the client-side. A description in the form of features would be a possible higher level, so that developers only have to define which features are required.

Another issue is the complexity of the generated web applications. We originally wanted to generate reusable web components. Unfortunately our generator evolved and its complexity makes our components difficult to reuse. A next step would therefore be to focus on reusable web components that can be partially used in other non-generated projects.

Finally, we have to deal with platform independence. Currently, our generators have been developed for Polymer 1.0. However, a lot has changed in version 2.0, so the generators are difficult to adapt. It remains to be clarified whether generators can be built that can operate independently of the platform used.

REFERENCES

Amundsen, M. (2017). *RESTful Web Clients: Enabling Reuse Through Hypermedia*. O'Reilly Media.

Beltramelli, T. (2017). pix2code: Generating Code from a Graphical User Interface Screenshot. *arXiv preprint arXiv:1705.07962*.

Bettini, L. (2013). *Implementing Domain-Specific Languages with Xtext and Xtend*. EBL-Schweitzer. Packt Publishing, Limited.

Cooney, D. (2014). Introduction to Web Components. https://www.w3.org/TR/2014/NOTE-components-intro-20140724/.

Darwen, H. and Date, C. J. (1995). The Third Manifesto. *SIGMOD Rec.*, 24(1):39–49.

Efftinge, S. (2014a). Xtend Documentation. http://www.eclipse.org/xtend/documentation.html.

Efftinge, S. (2014b). Xtext Documentation. http://www.eclipse.org/Xtext/documentation.html.

Evans, A. (2015). Polymer Project 1.0. https://www.polymer-project.org/1.0/start/.

Fertig, T. and Braun, P. (2015). Model-driven Testing of RESTful APIs. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15 Companion, pages 1497–1502, New York, NY, USA. ACM.

Fielding, R. (2000). *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine.

Fielding, R. (2008). REST APIs must be hyper-text driven. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.

Fowler, M. (2010). *Domain-Specific Languages*. Addison-Wesley Signature Series (Fowler). Pearson Education.

Haverbeke, M. (2011). *Eloquent JavaScript: A Modern Introduction to Programming*. No Starch Press Series. No Starch Press.

Jaber, M., Falcone, Y., Dak-Al-Bab, K., Abou-Jaoudeh, J., and El-Katerji, M. (2018). A high-level modeling language for the efficient design, implementation, and testing of Android applications. *International Journal on Software Tools for Technology Transfer*, 20(1):1–18.

Jacobson, I. (1992). *Object-oriented Software Engineering: A Use Case Driven Approach*. ACM Press Series. ACM Press.

Jakob, M., Schiller, O., Schwarz, H., and Kaiser, F. (2007). flashWeb: Graphical Modeling of Web Applications for Data Management. In *Tutorials, Posters, Panels and Industrial Contributions at the 26th International Conference on Conceptual Modeling - Volume 83*, ER '07, pages 59–64, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

Jovanovic, J. (2011). Designing User Interfaces For Business Web Applications. *Professional Web Design: The Best of Smashing Magazine*, 1:89–108.

Kelly, S. and Tolvanen, J. (2008). *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley - IEEE. Wiley.

Richardson, C., Rymer, J. R., Mines, C., Cullen, A., and Whittaker, D. (2014). New Development Platforms Emerge For Customer-Facing Applications: Firms Choose Low-Code Alternatives For Fast, Continuous, And Test-And-Learn Delivery. https://www.forrester.com/report/New+Development+Platforms+Emerge+For+CustomerFacing+Applications/-/E-RES113411.

Richardson, L. and Ruby, S. (2008). *RESTful Web Services*. O'Reilly Media.

Schreibmann, V. and Braun, P. (2015). Model-Driven Development of RESTful APIs. In *Proceedings of the 11th International Conference of Web Information Systems and Technologies*, pages 5–14. INSTICC, SciTePress.

Shimomura, T., Takahashi, M., Ikeda, K., and Mogami, Y. (2003). Web application generator by image-oriented design. *SIGSOFT Softw. Eng. Notes*, 28(2):14–.

Ulsamer, P., Fertig, T., and Braun, P. (2018). Feature-oriented Domain-specific Languages. In Riebisch, M., Huhn, M., Hungar, H., and Voss, S., editors, *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme XIV, Schloss Dagstuhl, Germany, 2018, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, pages 31–40. fortiss GmbH, München.

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.

Vu, H., Fertig, T., and Braun, P. (2017). Towards Model-driven Hypermedia Testing for RESTful Systems. In *Proceedings of the 13th International Conference on Web Information Systems and Technologies - Volume 1: WEBIST,*, pages 340–343. INSTICC, SciTePress.

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

Zuzak, I., Budiselic, I., and Delac, G. (2011). *Web Engineering: 11th International Conference, ICWE 2011, Paphos, Cyprus, June 20-24, 2011*, chapter Formal Modeling of RESTful Systems Using Finite-State Machines, pages 346–360. Springer Berlin Heidelberg.