Generating REST APIs for Workflow Engines with Process and Data Models

Dennis Pietruck and Ulrike Steffens

Hamburg University of Applied Sciences, Hamburg, Germany
{dennis.pietruck,ulrike.steffens}@haw-hamburg.de

Abstract. Workflow engines for automated execution of process models are typically run as back end systems and provide APIs for connecting software components. These connecting components reside on a different level of abstraction, using business- instead of process-specific concepts. To find a remedy, workflow engine APIs are frequently mapped to business-specific APIs specifically developed for this purpose. Modelling a business-specific interface for a workflow engine is a time consuming task.
In order to reduce development effort, this paper proposes a method to automatically generate domain-specific REST APIs based on both, process models and complementing resource information with only small extra modelling effort, thereby bridging the gap between process and data perspective. We furthermore present an architecture that can be used to implement the proposed method for any workflow engine which provides a REST API. The method promotes the continuous maintenance of process and resource information models. We introduce an initial prototype for the Camunda workflow engine, discuss limitations of the method and give an outlook for future work.

Key words: Resource Oriented Architectures, Business Process Management, Workflow Engine

1 Motivation

Workflow engines for automated execution of process models are typically run as back end systems and provide REST APIs for connecting software components. Using these APIs in a specific business context comes with some challenges. Starting from a short description of business process management on the one side and resource-oriented architectures on the other side, this section explains these challenges and motivates our proposed approach for an automated generation of business-oriented APIs for workflow engines.

1.1 Business Process Management and Workflow Engines

Many enterprises have adopted process management methodologies to improve the efficiency, reliability, and maturity of their processes. The discipline of business process management (BPM) deals with the whole life cycle of business

processes, comprising their initial conception / discovery, analysis, modelling, execution, and monitoring [1]. A number of standards for modelling business processes have been developed, not only aiming at documenting the modelled processes in an understandable manner but also at making models executable, hence calling for automation of business processes. One notation widely adopted by the industry is the Business Process Model and Notation (BPMN 2.0, here BPMN in short). Several workflow management systems (WFMS) for the execution of BPMN process models [2] have been conceived. Automating business process execution through WFMSs enables fast adaption of processes to new conditions. By describing the process logic in the model rather than in code, it can be ensured that processes follow the current specification.

WFMSs rely on BPMN models to control the process flow and to coordinate between process instances and their participants. Some examples of open source WFMSs are Camunda[1], Activity[2] and jBPM[3] which have some features and components in common. They provide, for example, a process modeller to design executable process models and a process management interface, which is a graphical interface for end users to control and to take part in process execution. The core component of these WFMSs is a process engine that executes process instances on the basis of process models provided in form of XML files. In order to manage the data required within the process instances, WFMSs usually employ the concept of process variables [3–5] which can be read and manipulated to share information between process activities and with components lying outside the process.

1.2 REST and Resource-Oriented Architecture

Despite the existing GUI, process engines are often embedded as backend services into existing applications or distributed scenarios. To couple an engine with external components, it usually provides a Java API that can be used to implement service tasks or control the flow of process instances. In addition to the Java API, most engines offer a remote REST interface. Representational state transfer (REST) is a widely adopted architectural style. Due to their simplicity, REST APIs are used in many use cases such as communication between internal services, communication with end-user devices or as an offering for third parties [6]. Since the Hypertext Transfer Protocol (HTTP) is based upon the REST principles, REST interfaces and clients are easy to implement with standard libraries of common programming languages, which are mostly equipped with HTTP server and client functionalities.

REST promotes properties such as scalability, loose coupling or intrinsic interoperability of web services, which all are required under the conditions of the World Wide Web. One central constraint that distinguishes REST from other architectural styles is the uniform interface constraint. This constraint

[1] https://camunda.com/
[2] https://www.activiti.org/
[3] https://www.jbpm.org/

among others requires that an individual resource is identified through a unique identifier, which is used in requests. The representation of a resource, that is returned to the client, is decoupled from the server's internal representation of this resource. Furthermore, the client itself can modify a resource, if it holds a representation of the resource and all attached metadata. Additionally, it is required that every message contains all necessary information to be processed. Lastly, the uniform interface constraint requires the use of hypermedia as the engine of application state (HATEOAS). To implement HATEOAS, a server enriches its response regarding one resource by adding links to other connected resources and the methods applicable to them. This enables automatic discovery of available resources for the client [7].

1.3 Problem Statement

Since the REST APIs provided by WFMSs are fine grained, using them leads to increased complexity and development effort for consuming clients. Querying or manipulating process data requires multiple API calls to access the needed resource. Since the API relies on BPM terminology, developers have to be familiar with both, BPM as well as business semantics and have to map concepts of both worlds, which is a time consuming activity [8] which possibly has to be repeated time and again when the enterprise's business processes undergo some changes.

The following example scenario might illustrate this problem. Figure 1 shows a simple manual check of an application which might e.g. be carried out in some insurance company. Let's suppose a developer working on some front end for clerks who will later carry out the application check is using the REST API of a process engine to implement the respective functionality. To do so, he/she would need to implement all of the following steps.

1. fetch the process instance that handles the considered application
2. fetch the task "decide application" that belongs to the process instance
3. set process variables that represent the decision of the user
4. set the task's state to "complete"

While a well designed REST API residing in the application domain would only require one step to update the state of a decision resource to realize this task, relying on the WFMS's REST API, the developer has to undergo several steps and has to match process concepts (e.g. task, process variable) with business concepts (e.g. decision).

To overcome this issue process engines are often used as an embedded library, which is then wrapped by an API that reflects the application domain. This approach moves the complexity of directly consuming the API of the engine to the wrapping service that is using the engine. In this way, the development effort is not completely avoided but only shifted. Furthermore, REST and BPMN have a different view on a system. While BPMN provides a process-centric view of a system, REST provides a view that focuses on the resources and their

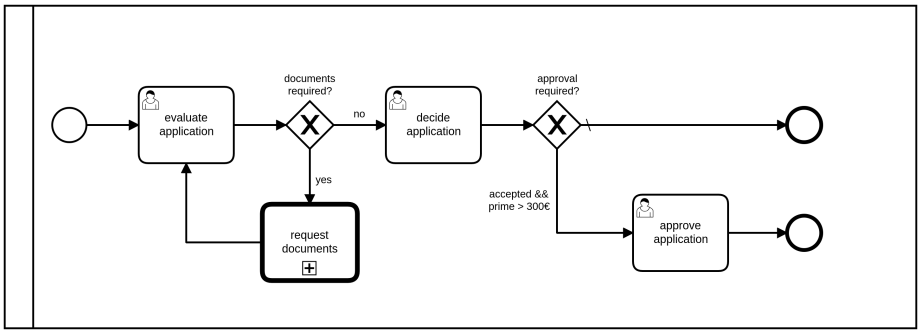

Fig. 1. Manual application check

associations [9]. Merging these views is a further effort in the development of a consumer friendly interface.

Looking at the challenges described above, automating the creation of a REST API for the process engine is an attractive approach. This work proposes a respective method that closes the gap between the different views and should reduce development effort by automatically generating an API that reflects the business domain, and uses a WFMS as the back end. The API is generated by using a data model which describes the REST resources and their relations, together with a BPMN process model. In addition to the reduced effort, the procedure promotes the continuous maintenance of data and process models.

1.4 Structure of the Paper

The remainder of this paper is structured as follows. Section 2 gives an overview on related work. To further motivate our approach of an automated generation of business-centric APIs, we describe how it is embedded into an overall application development process. Details of the approach are explained in section 4 and the development of a first prototype is presented in section 5. Finally, section 6 concludes this paper and gives an outlook on future work.

2 Related Work

In [9] a modeling approach for REST APIs is presented. This approach consists of several model layers describing the REST API in more detail layer by layer. It starts with a domain model from which resource models are derived. Based on the resource models and their associations a model for the resource URLs is deduced. This can then be used with the resource model to generate Java code that implements the API. A similar approach, but consisting of fewer model layers, is described in [10]. Here the eclipse modeling framework is used to generate Java code from resource models using the JAX-RS API. Although it is possible to describe the resources and the callable methods in the models, the approach

neglects the resource state in the model and cannot decide when which operation is available. Other approaches are using domain specific languages (DSL) instead of graphical models to generate the interface. In [11], for example, an approach is presented that uses a DSL and a description of the database as input to generate a REST interface. Here, again, the resource state cannot be embedded into a business process.

In addition to the approaches in which API code is generated, there are also some that generate an entire application. In [12] a procedure is presented which generates the interface code and the persistence layer as a NoSQL database. This solution increases developer productivity by creating a basic application that can be further customized.

Using the REST architectural style for business process applications has been research subject of several projects. Various research questions were examined. Pautasso proposes in [13] how to add REST resources to the BPMN specification. For this purpose, a new symbol is introduced that highlights a process, event, or task that is exposed as a resource. This principle is partially implemented by common process engines. [14] describes a way to access process information through a REST API. These approaches, however, do not decouple the interface from the underlying business process and knowledge on the process model as such is required. Since HATEOAS allows controlling the user interaction flow, it shouldn't be necessary to expose the process to the API consumer. A solution to this requirement will be presented in the following chapters of this paper.

Another question concerning BPMN and REST is how BPMN choreographies can be realised with REST interfaces. [15] and [16] introduce an organisational layer which is supposed to manage BPMN choreographies between different parties. A UML class diagram is used to describe the resources interchanged during the choreography. We choose a similar way of representing resources in our approach. Furthermore, [15] and [16] use the Object Constraint Language (OCL) to model the state of resources. This enables the creation of process agnostic interfaces. However, the focus of [15] and [16] is on creating interfaces for choreographies with different process participants. In this paper, we abstract from the communication with the process engine by introducing a generated domain-oriented interface. Moreover, the state of resources is bound to the state of tasks, which reduces the possible states and keeps the modelling effort low.

3 Embedding generated REST APIs in the application development workflow

Before we explain in detail how the REST APIs for business processes are generated, we provide a brief overview of how application development with a generated interface can be performed. For this purpose, we suggest a possible development workflow (see fig. 2) using the automatically generated interface. After the business process has been modelled and the input for the user tasks has been determined, developers model the application's resources. Process model

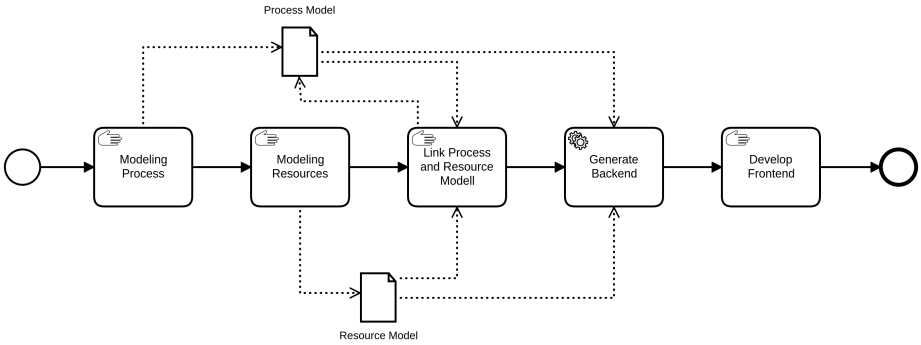

Fig. 2. Workflow for using generated APIs

and resource model are linked in the next step by annotating the process model with resource names. The API can then be generated with the annotated process model and the resource model. Finally the front end for the user tasks can be implemented. Process monitoring can be performed during process execution. Many process engines already offer standard functionalities for this purpose. If optimization opportunities such as bottlenecks are identified or compliance reasons require a restructuring of the process, the described steps can of course be repeated. Besides an adjustment of the process model, a change of the resource model is possible at any time.

4 Generating REST APIs with Process and Data Models

This section describes in detail how process models and resource models can be connected to automatically generate REST interfaces for business process applications. Figure 3 visualizes the concept of the approach. It can be roughly divided into two layers. The upper half shows the constituents of our approach, i.e. the models and model elements involved, as well as their interrelations. The lower half gives a dynamical perspective on how these constituents are instantiated, linked, and executed at runtime.

The process model and a UML class diagram modelling the REST resources are created during the first two activities of the workflow we introduced in section 3. Modelling the REST resources is described in more detail in section 4.1. The link between the models is created by connecting the process and tasks of the process model with the individual classes modelled in the class diagram. This connection is described in figure 3 as a composition between a process or some of the tasks and the classes. This connection closes the gap between resource semantics and process semantics. It indicates that the state of the process /task output is represented by the resource and can be manipulated by it. Further details about this connection can be found in section 4.2. Figure 3 also shows that the classes in the class diagram describe the resources and the corresponding URIs. The URIs are also important for the connection between process instances

and individual resources. Section 4.3 and 4.4 describe how the URIs for resources are generated and how the mapping between a process instance and a resource is established. The relationship between task executions and resources is described in section 4.5. It also explains how the task execution state is used to determine the state of a resource.

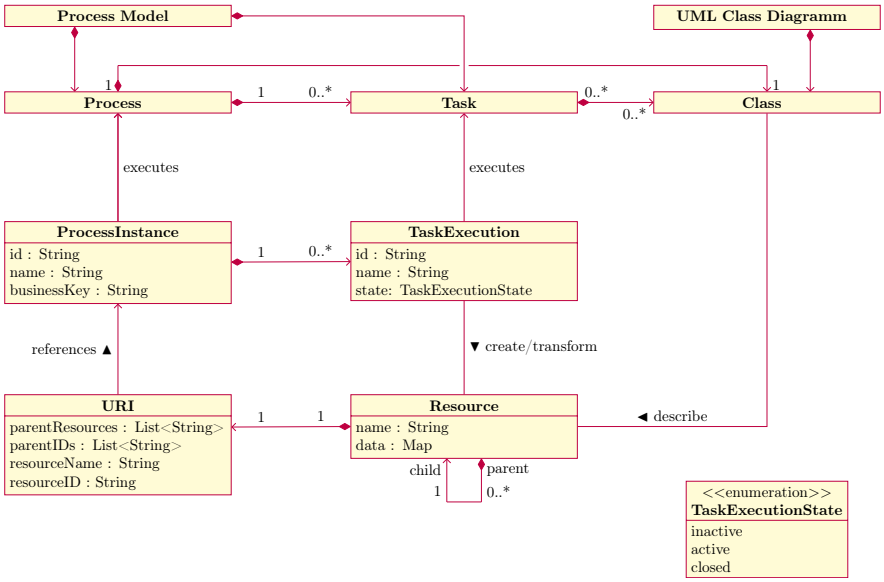

Fig. 3. Connecting REST Resources with Processes

4.1 Modelling Resources

As in other approaches dealing with the generation of APIs, UML class diagrams are used for the modelling of REST resources. This suits our purpose, as the attributes of the resources can be described as class properties. Furthermore, associations can be used for the modelling of the URL hierarchy. Figure 4 shows a simple resource model for the application check process we have already described in section 1. An application includes none to many documents and a decision whether the application has been accepted and what the insurance should cost. The decision also includes an approval. For simplification, the resource model can only use directed associations here and must already represent the URL tree. This simplification is chosen because this work only deals with the linking of the resource-oriented view with the process view of a system. For future versions of the tool which generates the interfaces we envision the use of domain models as in [9].

4.2 Connecting Resource and Process Models

To generate the interface, the business process for which the interface is generated must be linked to a resource. The idea behind this is that each business process processes or produces a resource. This concept is also reflected in common definitions of the term business process, such as e.g formulated by Davenport:

> In definitional terms, a process is simply a structured, measured set of activities designed to produce a specified output for a particular customer or market [17, p. 5].

The state of this output, be it a product or a service, can be described by the resource with which the process is annotated. This idea is also applicable to tasks. If a task is annotated with a resource, it means that this resource is being processed in this specific process step. The resources associated with the tasks of the process can be sub-products and services required for the execution of the business process, or the process resource itself which is modified in the business process.

Technically, the linkage between process or task and a resource can be described by an annotation in the process model. The BPMN2.0 allows the extension of process models by *extension elements*. To create a link between a process element like a task and a resource of the class diagram, BPMN extension elements can be used. An extension element is an XML element that can be a child of a regular element of the BPMN specification to augment a process model with custom data. Listing 1 shows a BPMN user task named "evaluate application", which is linked to the resource named "evaluation" by such a BPMN extension element.

Listing 1. BPMN user task with extension elements

```
<bpmn:userTask id="Activity_04bi4lp" name="evaluate␣
    ↪ application">
        <bpmn:extensionElements>
                <linkedResource name="evaluation"/>
        </bpmn:extensionElements>
        <bpmn:incoming>Flow_1dzcmml</bpmn:incoming>
        <bpmn:outgoing>Flow_0wx3he6</bpmn:outgoing>
</bpmn:userTask>
```

4.3 Generating URIs

As depicted in figure 3, a resource consists of a name and data defined by the class diagram. In addition, a resource has a URI that identifies it. A URI is generated as follows. Typically, the URI represents the relationships between different resources by using subresources. Although this is not mandatory, it can contribute to the comprehensibility of the API for developers. Therefore, our tool can also interpret the subresource relationships represented by URIs.

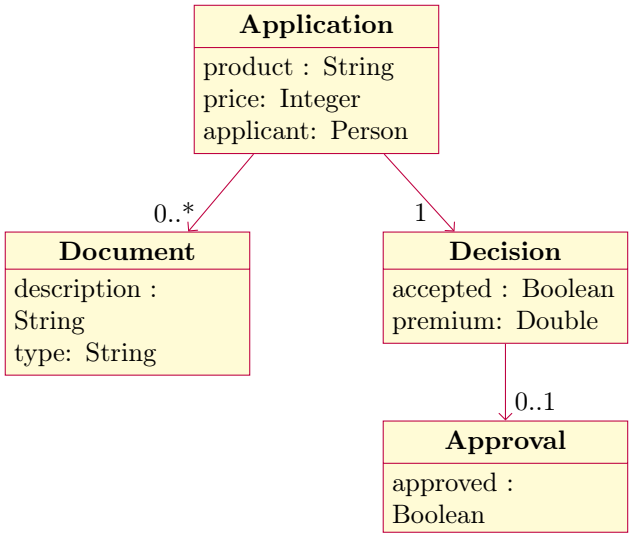

Fig. 4. Modelling the Resource Tree

The root resource of the API is always the resource which is linked to the process and is at the top of the modelled resource hierarchy. So, for our example, an "Application" resource modelled in figure 4 is identified by the URL */application/{applicationID}*. The order of the child resources is then formed by the associations of the class diagram. To identify the decision associated with an application, the URL */application/{applicationID}/decision* is used. The underlying resource "Approval" would then be identified with the URL */application/{applicationID}/decision/approval*. In addition to the direction of an association, the multiplicity of an association is also important for URI creation. For 1-to-1 relations, routes can be formed without an identifier of a subresource, because the subresource is already identified by the parent resource. Therefore, no further identifier is needed for the path to the "Decision" resource. For 1-to-N relations, resources of the same class have to be distinguished. For this purpose, the path to the subresource, which can exist multiple times, must be equipped with an identifier for the respective subresource. To address a certain document resource the path */application/{applicationID}/document/{documentID}* would be used. The URI class in figure 3 summarizes which information is required to generate a URI. The parentResources attribute is used to display the associations of the resource model. In addition to the identification of a specific resource, the parentIDs and resourceID attributes are also used to map resources to process instances. The mapping is explained in the following.

4.4 Mapping Instanciated Resources to Process Instances

After explaining how resources are identified in the REST API, it is explained how a resource can be assigned to a process instance. The assignment of a re-

source to a task alone is not sufficient, because in a typical application scenario several instances of a modelled business process are executed simultaneously. Therefore, a mapping from URI to process instance must be done to ensure that clients receive the data of the respective process instance. BPMN call activities start further process instances from a main process. These instances each have an ID of their own, so they cannot be assigned to a common parent process. A functionality that is offered by many workflow engines is the use of a business key. This key can be passed on to child processes that were started by call activities and thus enables different process instances to be assigned to their parent process.

Alternatively, process variables can be used. Here, data required for process execution can be added to process instances as process variables. If a WFMS does not offer business keys but variables, the key can be implemented by a variable itself. In our approach the business key is used to assign the root resource, i.e. the resource that is also linked to the process, to all associated process instances. Thus the URL */application/{applicationID}* for the application resource from figure 4 is equivalent to */application/{businessKey}*. This way of generating identifiers makes it easy to find every process instance in which the application resource is processed. For subresources with a 1-to-1 relationship, the business key is also sufficient as the only identifier. More difficult is the assignment of resources to tasks which have a 1-to-N relationship to their parents. As an initial simplification, we assume that 1-to-N relationships occur when the process execution loops through a subprocess several times. These subprocesses each have an ID of their own, which can be used as identifier for the resource that occurs multiple times. In this way, document resources from figure 4 are addressed with the URL */application/{businessKey}/document/{processInstanceID}*. The simplification for ID creation of resources with 1-to-N relationship excludes some use cases. These will be explained in the outlook of this paper.

4.5 Implementing HATEOAS

To implement HATEOAS it must be clear how the state of a resource is controlled. In our case the state of the resource should be determined by the business process. A method often suggested in the literature to model REST APIs are state machines. Linking activities to resources, delineates such a state machine. The BPMN2.0 specification also describes the life cycle of an activity as an automaton. For simplification, we restrict the state machine to the states *Inactive*, *Active* and *Finished*. These states can now be applied to the resource associated with the task. In our tool, a resource can only be changed if the respective activity is in the *Active* state. Read access is possible at any time. For the implementation of HATEOAS, a server response should contain links to parents and child resources, as well as the possible operations on the requested resource. These can be derived from the state of the annotated task. If a resource is linked to several activities, it can be processed if at least one annotated task is *Active*.

5 Prototypical Implementation

After showing how process models and resource models can be linked, this section presents a prototypical implementation. Therefore a generic modular architecture is introduced which can be implemented with different technologies. A goal of the architecture is that individual components can be replaced fast due to the generic core of the architecture. This allows changing the back end process engine as well as the HTTP library that receives the requests. Figure 5 visualizes the main components of the generic architecture.

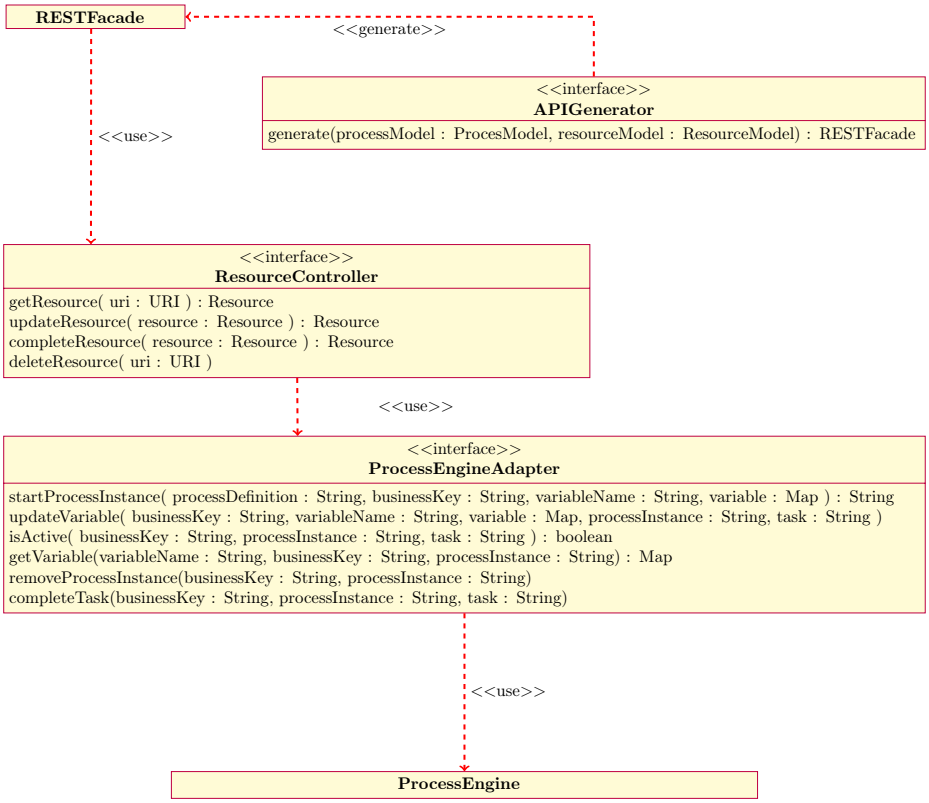

Fig. 5. Modular architecture for implementing Process Driven REST APIs

HTTP requests are accepted by the RESTFacade. The RESTFacade converts the request data into the required data types and calls the corresponding methods of the ResourceController. The facade is generated by the APIGenerator. This should be done at runtime without generating code. This is possible because a generic data structure is used for the representation of the resource. The APIGenerator only has to generate HTTP endpoints for each resource from the model. The corresponding method call to the ResourceController is then

the same for each resource, but with different parameters. These parameters are adapted by the generator for the respective resource. The ResourceController maps the resource logic described in the previous chapter. Here the conditions like the state of the tasks for a resource or the attributes sent with a resource are checked before a task is processed. Afterwards, the changes are passed on to the ProcessEngine using the methods offered by the ProcessEngineAdapter. The ProcessEngine is abstracted by the ProcessEngineAdapter. This adapter makes the required calls to a ProcessEngine and must therefore be implemented individually for each engine. Calls to the ProcessEngine that are required could be for example:

- start a new process instance
- check whether a task is active
- set a process variable
- complete a task
- read a process variable
- stop a process instance

The architecture for the prototype was implemented in Java with the Camunda Process Engine. The Spring MVC web framework was used as HTTP library. To create HTTP routes dynamically for each resource, spring router functions where used[4]. This allows for a functional description of router behaviour at runtime, which is required to generate the resource URIs and create method calls to the Resource Controller that are adapted to each resource. The resource model is implemented as a graph by the Java graph library JGraphT[5], which enables easy querying of children and parents of a resource. The resources are stored as vertices. The associations between resources are stored as edges of the graph. The edges also store the multiplicity of the association.

The Camunda Workflow Engine comes with its own modeling tool, which can be extended with custom forms. This was used in our implementation to add resources to a task or a process without having to make direct changes to the XML document. This ensures that the resource annotations are inserted at the right places in the document. The models in the Camunda WFMS are stored in BPMN2.0 compliant XML per default. Like BPMN, there is a standard data exchange format for describing models in UML. The XML Metadata Interchange standard specified by the OMG serves as an open format for UML models. Various modeling tools already offer export of models into this format. For the sample implementation of the REST API generator, a simple parser that can read UML class diagrams was developed.

[4] `https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html\#webmvc-fn`

[5] `https://jgrapht.org/`

6 Conclusion and Outlook

This paper has presented a method to automatically generate domain-oriented REST APIs for business processes. We have made a proposal of how to connect process and resource models. We have described both, the linking of models, as well as the linking of REST resources and process instance by URIs at runtime. For this purpose, tasks are used as state machines for resources to control the possible operations on a resource. In addition, we have introduced an initial prototypical implementation of the presented approach. For the implementation a modular layered architecture has been used, which supports the interchangeability of process engines and HTTP libraries. The ResourceCotroller, once developped, does not have to be exchanged when changing libraries. Finally, we have explained how the tool can be used to create a user interface based on business processes using the generated API.

As already described in chapter 4, for the development of the prototype we assumed that a resource occurs multiple times in a process, if loops in the process allow a repetition of the associated task. However, it is possible that in a loop an existing resource is revised instead of a new one being created. It is also possible that when a task is repeated in a loop, the user has the choice of creating a new instance of a resource or editing an existing one. In order to execute these cases in the process, either conventions for modeling the process must be defined or the models must be supplemented with additional annotations that describe the behavior of the resources more precisely.

Also, it must be further defined how a change of the models affects the runtime behavior of existing process instances. Some WFMS already offer strategies for the migration from an older process version to a newer one. Here it must be evaluated whether the tool for generating APIs is compatible with these strategies and whether these strategies are also applicable to changes in resource models. In this paper, the automatic implementation of interfaces has been presented for BPMN processes only. However, processes can also be modeled within other langauge, like e.g. CMMN. Although the symbols known from BPMN can be used to model CMMN models, a different automaton with different states and transitions is used to represent the life cycle of an activity. In short, it is possible that tasks do not necessarily have to be executed but are still available and can be aborted at any time. These characteristics, which are important for CMMN, should be taken into account when linking the state machine to the resource.

Since this work is the presentation of the basic concept and a prototype only, further extensions for the developed tool are possible. In this paper, the API was implemented for the development of front ends for user tasks. However, REST APIs do not necessarily have to be used by front end applications. BPMN also enables more than just modeling human interaction with the process. Message events and receive tasks are further elements in the business process that require an interface to WFMSs. Here, it is also interesting how the presented method can be integrated into process choreographies. The work of Nikaj [16] already suggests some means of implementing REST for process choreographies, which we might take up in future work.

Furthermore, an evaluation of our tool is important. For this purpose, it can be examined in future work how the generated API affects the client applications. In order to measure the impact on the client code, a case study can compare the client applications that use different interfaces. Surveys of the developers allow an evaluation of the developer experience for our tool. Software metrics such as coupling, cyclomatic complexity can be used for a quantitative comparison of the impact of the domain-oriented generated interfaces with the impact of the generic interfaces.

References

1. M. Dumas, M. La Rosa, J. Mendling, and H. A. Reijers, *Fundamentals of Business Process Management*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2018.
2. M. Geiger, S. Harrer, J. Lenhard, and G. Wirtz, "On the Evolution of BPMN 2.0 Support and Implementation," in *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. Oxford: IEEE, Mar. 2016, pp. 101–110.
3. *Process Variables | docs.camunda.org*. [Online]. Available: https://docs.camunda.org/manual/7.5/user-guide/process-engine/variables/
4. *Activiti User Guide*. [Online]. Available: https://www.activiti.org/userguide/\#apiVariables
5. *jBPM User Guide*. [Online]. Available: https://docs.jboss.org/jbpm/v4/userguide/html_single/\#variables
6. A. Gamez-Diaz, P. Fernandez, and A. Ruiz-Cortes, "An Analysis of RESTful APIs Offerings in the Industry," in *Service-Oriented Computing*, M. Maximilien, A. Vallecillo, J. Wang, and M. Oriol, Eds. Cham: Springer International Publishing, 2017, vol. 10601, pp. 589–604, series Title: Lecture Notes in Computer Science.
7. R. T. Fielding, "REST: architectural styles and the design of network-based software architectures," Doctoral dissertation, University of California, Irvine, 2000. [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
8. D. Pietruck, "Konzeption von fachlichen REST APIs für Prozessanwendungen," Hamburg, 2018. [Online]. Available: https://reposit.haw-hamburg.de/handle/20.500.12738/8501
9. F. Haupt, D. Karastoyanova, F. Leymann, and B. Schroth, "A Model-Driven Approach for REST Compliant Services," in *2014 IEEE International Conference on Web Services*. Anchorage, AK, USA: IEEE, Jun. 2014, pp. 129–136.
10. H. Ed-douibi, J. L. C. Izquierdo, A. Gómez, M. Tisi, and J. Cabot, "EMF-REST: generation of RESTful APIs from models," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, ser. SAC '16. Pisa, Italy: Association for Computing Machinery, Apr. 2016, pp. 1446–1453.
11. G. Jia-di and W. Zhi-li, "Modeling Language Design and Mapping Rules for REST Interface," in *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, Oct. 2019, pp. 1–6.
12. B. Wang, D. Rosenberg, and B. W. Boehm, "Rapid realization of executable domain models via automatic code generation," in *2017 IEEE 28th Annual Software Technology Conference (STC)*, Sep. 2017, pp. 1–6.
13. C. Pautasso, "BPMN for REST," in *Business Process Model and Notation*, R. Dijkman, J. Hofstetter, and J. Koehler, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 95, pp. 74–87.

14. X. Xu, L. Zhu, Y. Liu, and M. Staples, "Resource-Oriented Architecture for Business Processes," in *2008 15th Asia-Pacific Software Engineering Conference*, Dec. 2008, pp. 395–402.
15. A. Nikaj, M. Hewelt, and M. Weske, "Towards Implementing REST-Enabled Business Process Choreographies," in *Business Information Systems*, W. Abramowicz and A. Paschke, Eds. Cham: Springer International Publishing, 2018, vol. 320, pp. 223–235.
16. A. Nikaj, "REST-ChoreografienRestful choreographies," Ph.D. dissertation, Universität Potsdam, 2019. [Online]. Available: https://publishup.uni-potsdam.de/43890
17. T. H. Davenport, *Process innovation: reengineering work through information technology*. Boston, Mass: Harvard Business School Press, 1993.