UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Vishal Desai

Model driven engineering of Hypermedia REST applications

Master's Thesis (30 ECTS)

Supervisor(s): Luciano García-Bañuelos

Tartu 2016

Model driven engineering of Hypermedia REST applications

Abstract:

When we talk about REST, we eventually force our attention to applications that follow a standard infrastructure which can be easily connected to other applications on the internet. With standardization of exposed structure of an application. The REST application modelling can follow any norms or semantics and there may be many meta-models that one can use for modelling. Eventually, this limits the standardization. A meta-model determines the norms and standards of modelling. And the Language framework defines the code structure for an application. Models only plays a role in defining the content of the code. So consider a scenario where we have a predefined framework for coding an application and we have the model for it's REST endpoints, in that case we have the required structure of the code as well as the required content. We have all it takes to generate the basic skeleton of a REST application programming inter-face based on that particular framework. If we code this manually using any programming language, we are doing a lot of manual work and this is directly proportional to the number of entities and resources in an application. Work increases as the number goes up. This is a real problem. Many tools have already been developed for this approach, but they limit to basic REST model-view-controller. Unlike Basic REST, Hypermedia applications allow the end user to have no prior knowledge of the REST application programming interface but a starting url making the application more dynamic and secure. The objective of this research is to come up with a solid, feasible and efficient solution to take as input the Structural and Behavioural REST models of an application, and generate a skeleton of hypermedia REST application programming interface. What is needed is firstly, a target framework for generation of code. For starting the focus would be on Java with Spring boot framework and spring MVC structure. The scope of this research is limited to Java language only. Later on it could be possible to expand to other languages. The output generation part has to be done in a manner that it would allow in future to generate code for various languages. This would serve as guidelines for future work. Secondly, there is a need to know what kind of inputs and what type of models would be needed. REST modelling consists of two parts Structural modelling and Behavioural modelling. Structural modelling is a product of Class diagrams of an application. Behavioural modelling is a product of State chart of dynamic classes of the application. It may be possible to take into consideration other inputs as well. For which a thorough study has to be made because in practice the plain state chart and class diagrams are not sufficient enough to achieve the solution. A more stereotyped versions are required to completely achieve the solution. These would be the input for the system. These inputs need to be defined more clearly. Once these questions are answered then it would be possible to move further and answer the next questions. There is a need to know how to input these models. There are many ways by which we could input the required inputs to the tool. One idea is to generate a user inter-face where the User would create the model and the IDE would interpret it then generate the code. Other is to take as input the file for the model and generate code. With reference to some already available ideas and solutions, a thorough research will be carried out to come up with a perfect blend of ideas that would lead to a solution which would be implemented into a user friendly tool. With a meta-model we will have a standardized input mechanism. And with knowledge of output framework we would make it possible to interpret and generate code. By integrating the standardized input mechanism with the frame-work specific generation mechanism, it will be possible to auto-generate a hypermedia REST application. This would save hours of manual work. And this is a real solution be-cause it incorporates existing stages of Software development and eliminates waste work.

Keywords:

REST, hypermedia, code-generation

[Kommentaarid]

Lühikokkuvõte:

Võtmesõnad:

Table of Contents

1 Introduction

1.1 Motivation

Described earlier in brief that human effort in creating a basic structure of a REST application is directly proportional to the number of entities in the resource model of the application. When we come to a point where the application has more than five or six entities this work gets multiplied. This ultimately results in excess human effort.

In order to reduce this work, a proper automation procedure has to be established. We need to find patterns and make use of reusable code structure which could be applied to a handful of entity specific files.

The goal of this research is to find how a bunch of inputs taken from the user could help us gain enough information to automatically generate end code within a prescribed scope.

1.2 Problem Statement

If we are able to design a REST application programming interface using these models and then write code, then why not skip the step and auto generate skeleton in REST [1] style and then code in the details. We need to define concepts clearly and how these concepts will be mapped to UML diagrams.

With use of several models an application can be represented showing its REST [1] properties in connection with its other features. With use of such models we could provide information to the system which could interpret it and generate code. REST [1] consists of two parts Structural modelling and Behavioural modelling [2]. Structural modelling is a product of Class diagrams of an application. Behavioural modelling is a product of State chart of dynamic classes of the application. It may be possible to take into consideration other inputs as well. For which a thorough study has to be made because in practice the plain state chart and class diagrams are not sufficient enough to achieve the solution. A more stereotyped version is required to completely achieve the solution which would be the input for the system. The input needs to be defined more clearly.

The objective of this research is to come up with a solid, feasible and efficient solution to somehow auto generate the basic skeleton code from Structural and Behavioural models within the application. Thesis aims at answering the following questions.

- How will it be possible to auto generate a Skeleton code for a Spring hypermedia REST application?
    - What kind of inputs in general are required to write a Spring hypermedia code?
    - How will we be able to gather information from the user about the Structural and Behavioural aspects of the application?
    - How will we be able to use the information to provide the inputs for the code?

1.3 Contributions

Studying the current state of the art, it should be taken into account the limitations as well as the potential plus points of existing technology. Knowing the possibility for enhancement of any form of existing project creation procedure, this thesis would aim at achieving some step by step results. The following would qualify as the benchmark results from this research,

- Identify information required from user when creating a REST application.
- Come up with a suitable meta-model that incorporates behavioural as well as structural aspects and to ensure that this information can be taken as input from user while modelling.
- Create a generation tool that would take this input that we define and would generate the skeleton code as the output.

2 Background

2.1 Common Words and Concepts

Before moving further into the paper one must understand some phrases and their meanings. Some of these are concepts are essential to the understanding of the research and the further sections of this paper.

2.1.1 REST

Representational State Transfer [1]. Refers to a Software Architectural style that follows guidelines on the footpath of Hypertext transfer protocol [1] making it currently the most favoured Web architectural styles for modern distributed systems and web applications.

- Web services and Resource Oriented Architecture

  Restful web services by Richardson et al [3] states the difference between REST and Resource oriented architecture. Though it might sound the same but REST is a set of guidelines and Research oriented architecture is much more.

  Every organisation and every service can have it's own architecture. Which would definitely be RESTful if the guidelines are followed but they all may not be the same. There is a great variety in which one could design the architecture of the web service using the REST guidelines and which would differ from others.

  In all Resource Oriented Architecture [3] is much more than REST itself. It is the architectural design of a service based on individual perception and understanding of REST concepts.

  Every Web service providing RESTful services with it's own Resource Oriented Architecture has but some common REST concepts.

- Resource

  Fielding [1] in his Dissertation describes a Resource as any object or item or information that can be named. A dog or cat or today's weather would qualify as resource for E.g.

- Resource Identifier and URI

  And every resource is uniquely identified by a Resource Identifier [1]. The service that involves this resource would be reached by the Uniform Resource Identifier (URI) [1] of the resource which for a single unique resource along with it's Resource Identifier is also unique. This enables service provision directly related to the resource involved.

- REST API design

  Masse et al REST API design rulebook [4] provides some very useful guidelines to designing the application programming interface and the URIs.

Every resource as we know is identified by a Resource Identifier and addressed using a Uniform Resource Identifier [1]. On this URI we can perform a set of basic operations. As per the HTTP basics the methods such as GET, PUT, POST, DELETE [5] can be applied while addressing an URI.

- Forward slash '/'

  The design of a REST api follows a hierarchical structure i.e. every time a '/' is seen it would be perceived as a hierarchical division. Very similar to the HTTP addresses a URI of the domain will serve as the highest level of hierarchy and following would be next levels.

- Hyphen '-'

  Hyphens may be used to make the URI easy to interpret and for better user readability. A hyphen could be used to link two words that in general language terms would be separated by a single space but from the application programming interface terms are collectively involved in making a meaningful concept.

- Underscore '_'

  Underscore should not be used in the URI design. [4] Because of a very common universally accepted practice of providing blue text colour and underlining font for a link, underscore would make the link not so easy to interpret.

- Use of lowercase letters only

  Use of uppercase letters would be preferred as capital letters could sometimes cause problems.

- PUT [5] method must be used to update a resource

  When a resource is ready to be updated the preferred HTTP method to invoke would be PUT.

- POST [5] must be used to create a new resource

  When a new resource is meant to be created then the preferred HTTP method to invoke is a POST method.

- DELETE [5] must be used to remove a resource

  When a resource is meant to be deleted the preferred HTTP method to invoke is a DELETE method.

- GET [5] must be used to query a resource

When a resource is meant to be queried then the preferred HTTP method to invoke is a GET method.

- Container

  The container is a collection of a known resource. It is basically the entire set of elements that belong to a particular resource type [4].

- Resource

  The single element of the container identified by a Resource Identifier. [1]

- Hierarchy

  The hierarchy of a URI is maintained by use of '/'. Every resource will have a container class that holds multiple instances of the same resource. This container is the first level of hierarchy for that resource. We would put a '/' followed by the resource identifier that would identify for us a unique element from the container and that would be our target resource. Further hierarchy may be applied with use of '/' as the designer sees fit. [1]

- Pathvariables

  Parameters that serve a purpose in both the controller method invoked as well as the URI of the resource are the path variables. The resource identifier is the most common path variable as it serves a purpose in identification of a resource in the controller as well as the URI of that resource. [6]

- Path parameters

  Parameters that do not play an important role in the URI of the resource but play a major role in the controller method invoked qualify as path parameters. These are most commonly some search criteria for queries on a particular resource. Could be used otherwise as well if seen fit by the designer. [6]

2.1.2 HATEOAS

Hypermedia as the Engine of Application State [6] refers to a web application framework by Spring [7] which provides ability to build Java based REST [1] applications with hypermedia [8] support.

2.1.3 Resource

In terms of a REST [1] application, a resource is anything that can be accessed with a Uniform Resource Identifier (URI) [1].

2.1.4 Meta-model

A higher level model that defines "the structure and meaning" [2] of the model itself. According to Silvia Schreirer in 'Modeling Restful applications' [2] a meta-model is responsible for the syntax of the model, and plays a vital role in model driven approaches.

2.1.5 Application Programming Interface (API)

A complete representation of a software component enlisting it's methods, properties, hierarchy, inheritance and visibility.

2.1.6 Domain specific language (DSL)

A problem oriented programming language that consists of abstractions and notations specific towards solution to the problem rather than a generic form. [9]

2.1.7 XML [10]

Extensible mark-up language is a flexible hierarchical text format which is used widely as a standard for data transmission payload format.

2.1.8 JSON [11]

JavaScript Object Notion is a data transmission and interchange format widely used owing to it's simple parse-able format which suits use with a variety of programming languages.

2.1.9 JVM [12]

Java virtual machine is a layer of component technology that plays a middleman role in between the Java language and the host machine and operating system. This allows Java language to be machine and operating system independent.

2.2 Enterprise System Integration (ESI) methodology

The Institute of Computer Science of the University of Tartu has a course called Enterprise Systems Integration [13] which deals with providing students a good foundation to work and research on REST [1] services with the Spring framework [7]. As an alumni of this course, I find it interesting to further my knowledge provided by the course by incorporating the methodologies taught into this research because of it's very close relation with my research topic. Following are some concepts that incorporate the overall methodology of Enterprise Systems Integration.

2.2.1 Scenario for Analysis

The following is a scenario obtained from the University of Tartu's web portal for the above mentioned course [13]. It has been used exclusively in courses at the University.

The scenario chosen for this research is a subset of the bigger plant hire scenario in which there are two systems. For the purpose of this paper we will focus on only one of the systems.

- **The Plant Hire Scenario** [13]

The scenario we would use for analysis would be a simple one based on online renting of equipment. *Rentit* is a company that rents heavy machinery like cranes, tractors etc. known as *Plants*, to its customers via an online renting portal.

The portal provides a catalogue of plants to view. A *Plant* has a name, company name, id and price per day. After careful review of the catalogue, the order can be placed by the customer in the form of a *PlantHireRequest* which contains the id of the *Plant*, the date of hire and date of return.

A *PurchaseOrder* is created from the incoming *PlantHireRequest* and is stored in the database. The *PurchaseOrder* contains the start date which corresponds to the date of hire in plant hire request, the end date which corresponds to the date of return of the *PlantHireRequest*, the *Plant* id and the total cost excluding additional costs. The status of the purchase order will be open once created. An executive will review the order and confirm or deny it. If it is confirmed then the delivery is made. If it is denied, then an email is sent to the customer to notify of the decision and the reason after which the customer can send a fresh plant hire request.

A purchase order can be cancelled or updated within 18 hours prior to the date of hire. In case of updating the purchase order, the new start date should have more than 18 hours of difference from the time of update and the plant has to be available within the newly set period of hire.

After the delivery is made, it can be accepted by the customer or rejected. In case of rejection the plant is brought back and based on the customer's reviews, the *PurchaseOrder* could be opened again and a fresh delivery be made. Once a delivery is made and the plant returned, the purchase order is marked as closed and an *Invoice* is created.

The *Invoice* which is unpaid is sent to the customer's email. The customer will review the *Invoice*, and if the customer accepts the invoice, then the customer can make a payment. If the payment is made then the invoice is marked as paid. If the invoice is unpaid for more than 7 days then a reminder email is sent to the customer.

2.2.2 Analysis and design

The phase of analysis will undergo process of identification of resources within our scenario text. During analysis it is essential to understand every sentence within the scenario text. This includes grammar of the language it is written in. The structuring of a sentence in the written language plays a major role in identification process. The methodology of ESI states some ground rules for identification.

- Nouns

  Every noun encountered in our scenario will qualify as a potential *resource, entity or property*. The importance of a noun found in the text is also essential to come to a decision. Finding a noun does not necessarily mean that this noun qualifies as a resource but it sure does qualify it. If semantically the noun is found important then it can be considered a resource. Nouns can be spotted differently in different languages. In English language for e.g. a noun could be spotted when seen to be used by a definite or an indefinite article like 'a', 'an', 'the', 'that' etc.

E.g. in the scenario above, ‘A PurchaseOrder’ and ‘the database’ both qualify as potential *resources*. But semantically only PurchaseOrder can be a resource. Also *startDate* and *endDate* qualify as properties of a resource, which is *PurchaseOrder* etc.

- ‘being’ verb

  The form of the verb ‘being’ within the text usually helps us identify state of a resource. Once we have identified the nouns that semantically qualify them as *resources* or *entities,* the verb form of ‘being’ that describe them can potentially give us information about the state of the resource. Once again, semantically it should important. If the verb is in its present form i.e. ‘is’ then it would indentify the current state of the resource in the context of the sentence. The future forms like ‘become’, ‘will be’, ‘would be’, or simply ‘be’ would identify the future state(s) of the resource from it’s current state.
  E.g. in the scenario, ‘the Invoice which is unpaid’ implies that the current state of the invoice is *UNPAID* etc.

- Prepositions

  Prepositions in general helps us identify relations between two *resources* or between *resources* and *actions*. Once we have identified the resources, we can analyse them in semantically to find potential relations between two *entities* or *resources.*

  Semantically one must analyse the prepositions used to indentify the meaning. The preposition ‘of’ for instance would imply the content of a resource.
  E.g. in the scenario, ‘The status of the purchase order’ implies that PurchaseOrder contains a Status.

- Verbs

  Verbs help us in identifying actions. Actions semantically important such as ‘PurchaseOrder can be accepted by the customer or rejected’ implies that ‘accepting’ is an action.
  Verbs usually help us identify functions within our application. E.g. *accept* and *reject* would be two functions on the resource *PurchaseOrder* etc.

2.2.3 Domain modelling

Classic Domain model can be useful in distinguishing *entities* from *resources*. *Entities* end up becoming *resources* but in many cases there are resources that may not be an *entity* in your application. Search related resources or the resources that only carry information can qualify as such resources. These resources will not be present on the domain model.

Domain model will consist of anything that will be tangible to lower-layer persistence of the application which play a role in providing data to the higher level application programming interface (API). Usually domain model will begin with the data model of the application which consists of all the *entities.* When refined into the containing only the most important entities which will play an active role in providing data to the higher level API, it becomes a domain model. The entities which provide this data will remain while other will not be a part of the domain model.

2.2.4 Resource modelling

Resource modelling is the next step towards designing a REST application programming interface (API). The resource model would usually begin with the domain model. The domain model will consist of data which can be transformed into data required for the higher level API. It can happen that in some situations the entities that exist in the domain model are not present in the resource model. This is because thought the entities play a role in determining the outcome of the resource model the entities themselves are not present in the Resource model, because they are not available at the higher API level.

The key entities that remain in the resource model will become the *resources*. The collections that remain will become *containers*. Every single element of this container will become an individual *resource*. Associations from domain model will become *references (ref)*. Every reference will reflect on the API as a *relation (rel)* in the URL.

2.2.5 Behavioural modelling (State diagram)

Key resources in the resource model will have multiple states. Such are the resources whose behaviour changes as the application reaches the new stage in the life cycle. Every resource who behaviour changes will have a status identifier within it. This identifier will determine the current state of the *resource*. Every state will have transitions. A transition can be from one state to another and can be triggered by an HTTP request on that resource.

2.3 Overview of tools

2.3.1 Domain Specific language (DSL)

A DSL is a problem oriented programming language that consists of abstractions and notations specific towards solution to the problem rather than a generic form. [9] This aims at solving the problem at the program level. These are usually smaller languages with a very small scope.

With modern programming languages following a strict syntax along with a vast base, sometimes there arises a need to have something smaller, that existing languages do not provide free from complexity. With only the handful of required parsing and linking mechanisms that one needs for the purpose of fulfilling the requirements of the application, a DSL plays an important role in such cases where one can incorporate self-designed rules and syntax to accomplish a particular task.

Later in the paper, a detailed explanation would be given regarding creation of a DSL and it's syntax in section 4.

2.3.2 Xtext [14]

Xtext is an eclipse [15] based Domain specific language creating tool that allows you to create your own syntax. It also provides the possibility to generate files from the input information provided. This is useful in adding the hypermedia [8] support for a self-designed DSL. Along with that it could also be useful for generating a UML [16] based model input because it supports other eclipse based tools like Ecore and Ecore model [17]. Xtext is extensively used in this research. A detailed explanation on it's usage will be provided later in the paper in section 5.

2.4 Survey of Existing Technology

This study has been carried out in order to identify any existing technology that may be following the guidelines or the same objective. The following are the technologies that were studied to get an idea of the current state of the art.

2.4.1 Apiary IO [18]

This is a tool that allows users to create a mock REST [1] application programming interface before beginning the coding. This is useful as it allows us to perform integration tests with use of a mock service but does not generate code.

2.4.2 Swagger [19]

This allows us to have a very neat and clean representation of our REST [1] application. It also allows us to test our application programming interface endpoints with a very friendly user interface. But it does not have a support for Spring's hypermedia framework [8] i.e. HATEOAS [6].

2.4.3 RAML [20]

RAML is a language that can be used to model a REST [1] application. It has a variety of features and a unique syntax that makes representing a REST [1] application understandable. The shortcoming of RAML [20] is that it is limited to modelling Structural Aspects [2] of a REST [1] application. The behavioural aspects [2] like Hyperlinks are not supported. In modern REST frameworks like Spring's hypermedia [6], defining Links between resources plays a major role.

2.4.4 RestUnited [21]

RestUnited is another REST [1] application programming interface generator that allows end user to give resource [1] information. It can generate the skeleton in a number of languages. This is limited to Structural aspects [2] of an application. There is no hypermedia [8] support in RestUnited.

2.4.5 Restlet Studio [22]

Restlet Studio is a REST [1] application programming interface generator with a Web interface. It allows user to create a REST [1] application and specify resources [1] along with hyperlinks [8]. This works well as it can generate the code we need. But it does not follow a model driven approach [23]. In the sense of modelling, Restlet studio does not offer a modelling methodology. The user is expected to have a model ready and put the data into the interface step by step. This in turn does not reduce manual work. This is because the modelling is expected to be already achieved, and in the end putting information into a web interface is an extra work nevertheless.

2.5 Shortcomings of Existing technology

After thorough research on the existing technology it can be concluded that almost all of them do not have proper support for Spring HATEOAS [6] framework to develop a RESTful [3] application. Behavioural aspects are covered by some of them but with different frameworks. As we focus on Spring HATEOAS [6] we are in need of something that supports it. Some of the technologies are really good for use with Structural aspects [2] of a REST application but they have limited support for behavioural aspects [24]. The following shows survey of the existing technology and shortcomings.

Name	Pros	Cons
Apriary	• Useful to test mocks for REST api.	• Does not generate code.
Swagger	• Good documentation and testing of API. Code generation available.	• Does not support behavioural aspects.
RAML	• Start to End designing of API with documentation.	• Does not generate code. • Does not incorporate behavioural aspects
RestUnited	• Can generate code in several languages.	• Does not incorporate behavioural aspects. • Not free.
Restlet Studio	• Provides a friendly user interface.	• Does not incorporate behavioural aspects. • Does not provide a model driven approach.

Table 1. Shortcomings of Existing technologies

3 Contribution

This section will bring light into how the research will be carried out and how the existing State of the Art would play a role.

3.1 Scope

By having a clear idea of the language and framework that would be used for end code, we can have a good idea on how the end code would look like. For that we need to clearly define the scope for the research. Using Spring boot application reference guide [25] we can clearly define the structure of the code and naming conventions.

The scope of this research would be limited to creating a REST [1] application using the following technologies,

- Java 7+ [26]
- HATEOAS [6]
- Spring boot application and structure [25]
- PostgreSQL database [27]

3.2 Research method

We have identified certain technologies that could help us in achieving the goals as well as some related work that helps us in defining a proper meta-model. Along with the technologies we have also analysed potential drawbacks or shortcomings of the same.

In order to achieve the goal we have followed the following steps,

3.2.1 Identify information required from user when creating a REST [1] application.

To know what information is required from the user, it is important to know what kind of inputs are required during the coding. For this it was necessary to write the code manually in accordance with Spring boot application guidelines [25] to understand the inputs.

After coding and careful examination of the end code the following files were identified to be related to every resource in our REST [1] application.

Files for Resource ‘PurchaseOrder‘	**Information required**
PurchaseOrderResource.java	*@XmlRootElement name* Fields with *types* *Cardinalities* if applicable

PurchaseOrderRestControl-ler.java	*@RequestMapping value* for Class Functions with *Mapping* and HTTP *request type* *Parameters* and *Response Status* for functions
PurchaseOrderStatus.java	States *enumerations*
PurchaseOrderResourceAssem-bler.java	*Transition* between states HTTP *method type*, *output* function, output *mapping* for Transitions

Table 2. Files with expected information from User

3.2.2 Come up with a suitable meta-model [2] to ensure that this information can be taken as input from user while modelling.

Now that we have identified our Inputs from the user, we need to come up with a meta-model with the help of three chosen papers.

The study on the following papers have been conducted in order to come up with a suitable Meta-model to model a REST application. The objective is to study existing meta-models and to come up with a meta-model that ensures that all required inputs are incorporated while modelling a Spring HATEOAS [6] application.

- **Modeling RESTful applications** [2] **by Silvia Schreier**

This paper provides a very simple approach to Modelling RESTful application by use of ECore meta-model from Eclipse.

A very simple example of google picasa model is provided with related concepts. The paper addresses some core REST concepts like Resource and Resources and Resource types, States and Transitions, links and conditions etc.

These concepts used in the meta-model can be linked to a RESTful api and automation can be performed. The concepts of behavioural aspects of a REST application are thoroughly specified in this paper.

- **Towards a Model-Driven Process for Designing ReSTful Web Services** [23] **by Selonen et al**

This paper provides a step by step approach to developing RESTful services. The paper addresses structural as well as behavioral modeling of an application. The example used in this paper is an airline booking system and structural and behavioral models are provided.

This paper offers an approach to conceptualise Domain specific information and Resource oriented information within the common boundaries of a single structure model. The

behavioral cononicalisation model provides enough information about the application in terms of addressing and data holders. In this paper instead of using state chart model approach for behavior modelling they use behavioral canonicalisation which instead of perceiving the application as in many states, only perceives addressees and their bystanders which hold information and the actions that an addressee will perform.

Some important concepts provided in this paper include Addressees and Bystanders, refs and owns, listeners, qualifiers and contents, Create-Inspect-Replace-Remove, Containers and items. These can be mapped to RESTful concepts and would help in automation process. The analysis has been presented in the further sections.

- **Modeling Behavioral RESTful Web Service Interfaces in UML** [24] **by Porres et al**

This paper provides yet another approach to model-driven development where they address the structural as well as behavioral aspects of a REST application. This paper uses a simple Hotel reservation scenario to explain the approach.

The structural modelling in this paper gives a very concrete idea about the URIs of the REST api. A strict hierarchy is maintained and that makes it very easy to determine the resource paths.

The structural modelling here is termed as Conceptual model that has a collection and each collection has its element that can be addressed through the collection itself, maintaining hierarchy, with is resource identifier. Hence after the end of the conceptual modelling of an application we end starting from the system that serves as the root of hierarchy and follow the links. Every link ends with a new addition to the current URI. So as we keep moving down the hierarchy the model itself dictates the paths to the resources that serves as the URI in RESTful terms.

The behavioral model follows State chart based approach where the system is conceptualised as a bunch of states. Every state has a precondition and a postcondition. And then there is a transition with action and trigger. The concepts used in this paper are very useful in automating RESTful application. The analysis has been presented further in this paper.

Following shows the analysis of the three papers in accordance to our requirements identified above.

Required Info	**Selonen** [23]	**Schreirer** [2]	**Porres** [24]
Cardinality	yes	no	yes
Container Mapping values	yes	yes	yes
Function Mapping values	yes	yes	yes
Parameters	yes	yes	no

Responses	no	no	no
Transition Mapping values	no	no	yes
Function for transition	yes	yes	no

Table 3. Shortcomings of Review papers

From the above analysis it can be noted that there is some or the other missing requirement in the existing meta-models. So there is a need to make a new meta-model that would incorporate all the above required information.

3.2.3 Create a generation tool that would take the models as input and would generate the code as the output.

In this step we would make a Domain specific language using Xtext [14] that would follow the norms from the meta-model [2] we come up with based on the above analysis. With that we would use the generator functionality of Xtext [14] to generate the end code.

This step must also ensure that the information taken from user will be taken in a very user friendly way. The user should be able to write the information in a syntactical fashion and with minimum typing.

There could also be a possibility to bring in diagrammatic input methods, but this would be a secondary priority and a potential future work. The priority in this research would be to come up with a syntax that would allow the user to input the information with minimum repetition of keyboard typed words.

4 Information Input from the User

This section will deal with the detailed information on what input is required from the user and how we intend to achieve it.

As discussed earlier in this paper the scope of this research will limit to a Spring boot application [25]. Since many other frameworks exist for creating a Rest [1] application, this may serve as a limitation. But this limitation is only for the generation part. As far as the inputs from User is concerned it would be clear that User would be expected to give a certain number of inputs. This section will help you understand the inputs more clearly and their use in the final code.

For doing the research we will focus on a particular scenario, the Plant hire scenario [13], handpicked to contain several situations of interest. Refer to section 2.2.1.

4.1 Information from User

The most important 'to know' of this research is the inputs required from the user. To know that we need to know firstly what information is required to code a spring boot rest application [25] with hypermedia [6] Links.

In order to achieve this we need to first have a look at the end code. Knowing the end code will help us understand the resource specific entries in each class that we write in java.

As part of an attempt towards a broader understanding, some references in this section have been made to section 4.3 of this paper.

The above scenario would be modelled into the following,

4.1.1 Files

As per observation some files are common to all resources in the resource model. These files have a pattern and follow a similar process of coding. These files could be auto generated.

Now from the paper point of view, we would use the resource *PurchaseOrder* from the above Plant hire scenario [13], and analyse the files we need to code for this resource.

4.1.1.1 ExtendedLink.java

This file is an extension of Spring's [7] *Link* class. It provides functions to use and configure a hyperlink on a resource.

It provides an additional feature, that of having knowledge about the HTTP [1] method type that the link is valid for. This is done while instantiating the class as part of the constructor.

No information is required from the user to generate this file. This file has already been written and has no resource specific information within it.

4.1.1.2 ResourceSupport.java

This file is an extension of Spring's [7] *ResourceSupport* class. It provides necessary functions to add and manipulate hyperlinks [6], those already provided by spring and in addition it also provides a list of all hyperlinks [6] available on that resource at a particular state.

So at a given point in time during the lifecycle of the application, the resource in question can have a bunch of hyperlinks [6], which can change as the state of the resource changes.

No information is required from the user to generate this file. This file has no resource specific information.

4.1.1.3 PurchaseOrderResource.java

This file is the resource file in terms of Spring Boot application [25]. This is a simple java class with an annotation *XMLRootElement* [7] with *name* attribute which gives it the name for xml [10] or json [11] conversion.

This class extends *ResourceSupport* class described above. Extending resource support will allow the class to inherit some useful functions and features that would allow it to swiftly create and manipulate hyperlinks [6].

This class will be required for both, resources that originate at the application or 'internal resources' as well as resources that do not originate at the application or 'external resources'. These resource types would be explained further in the paper in section 4.3.

This class will be annotated with *@XMLRootElement* with attribute *name* which would bear the name of the resource. The same name would be used to represent the resource in xml [10] and json [11] formats. Along with this it would contain functions and datatypes along with their getters and setters. We would use the identity, *id* as a Long as per Spring boot guidelines. [25]

Resource specific information in this file is as follows,

- **Name of the file** (prefix to *Resource.java*)

  In the example, the name of the file is *PurchaseOrderResource.java* where the word *PurchaseOrder* comes from the name of the resource. This is so because it will make it easier to read and understand as well as easy to locate this file once all the files are created.

- **Name for the Class** (prefix to *Resource*)

  Could be named anything but once again in order to make things simpler, we would name the class the same as that of the file followed by 'Resource'.

- **Name for the xml/json** (value of the *name* attribute of *@XMLRootElement*)

  Every resource represented by a json [11] or xml [10] would have a root element name. This allows the recipient application to differentiate between different resources. We would name it the same as that given as the name of the class, but in lower case. This is to follow the naming conventions. [25]

- **Datatypes** (JVM [12] data types to be declared as attributes of this class)

  Java [26] data types can be declared as attributes of the class. These would serve as property fields of the resource. The values of these attributes will serve as the data that would be transferred by this resource. Every field will also have a getter function to get the value and a setter function to set a new value to the field.

- **Internal Datatypes** (other resources of the application that are contained by this resource)

  Any already defined resources can serve as data types of another resource depending on the resource model of the application. This means that one resource can contain another. In the Plant hire scenario [13] we observe that a *PurchaseOrderResource* contains *PlantResource* etc.

4.1.1.4 PurchaseOrder.java

This is the classic entity class in the model-view-controller world for a *PurchaseOrder*. This will have an annotation *@Entity* [7] which will induce the features of persistence. This class will serve as the domain model representation of the resource. This class will only be required for resources that originate at the application or 'internal resources', see section 4.3. This would be, same as the Resource file, containing datatypes with getters and setters. Again we would have identity, *id* as a Long. This class will only be needed for the resources that are native to the application.

From the Plant hire scenario [13] above, the *PlantHireRequest* will not have a *PlantHireRequest.java* class, because it does not originate at the application, but originates at the client and comes to the application as a payload in the body of the request. But, a *Plant* will have a *Plant.java* class, because it originates at the application and is sent to client along with the response to the request by the client.

Resource specific information in this file is as follows,

- **Name of the file**

  In our scenario, we have a *PurchaseOrder.java* which serves as an Entity [7] class for the resource defined. The name of the class has to be the name of the resource in order to make things simpler as well as for better understanding of the code structure in the future after it has been generated. I will make it easier to categorise the file and associate it with the resource.

- **Name for the Class**

  Would be the same as the name of the file to make things simpler. Also having the name same as that of the prefix name of the resource file will make it easier to locate after the class has been defined.

- **Datatypes** (JVM [12] data types to be declared as attributes of this class)

  Java data types can be declared as attributes of the class. These would serve as fields of the entity. In most cases the domain model will differ from the resource model and the fields of the resource will be different from the fields of that of the entity, but to begin with we will assume that the fields are same and later on it can be altered manually by the user as the software development cycle proceeds.

- **Entity Datatypes** (other Entities [7] of the application that are contained by this entity)

Any already defined entities can serve as data types of another entity depending on the domain model of the application. This will include cardinality by means of an annotation.

Eg, In the Plant hire scenario [13], we have a *PurchaseOrder* having one to one association with *Plant*. We can use the annotation *@OneToOne* before the declaration of the attribute *Plant*.

4.1.1.5 PurchaseOrderRestController.java

This will be the class that defines all the rest endpoints [6]. It will contain all the functions that are available as REST [1] services. This will be required for 'internal resources', see section 4.3.

This class will have the annotation *@RestController* [6] and for every specific function it may have the annotation *@RequestMapping* [6]. This annotation will have three mainly used attributes, *method* which will inform spring about the HTTP [1] method type that would trigger this function, *value* which will inform spring about what relation or uri [1] path this function will occupy in the application and it's endpoints and finally, *produces* which will inform spring about the media type of the response, xml [10] or json [11] etc.

In this class the functions usually have a direct access to the database with the use of repositories. Hence every function will perform some operation on the database hence we need an autowired repository for the resource in question. The *@Autowired* [7] annotation will allow the repository to be used in the class.

The behavioural aspects [2] of a resource also play a vital role in the coding of this class. According to the behaviour of the resource, different functions need to be placed here in this class to make change to the state of the resource. This adds up to the complexity of coding this class, as the states of the resource and their transitions have to be known before we can write this class.

For e.g. in the Plant hire scenario [13] above, *PurchaseOrderResource* has different states like *OPEN* or *REJECTED* etc. So depending on the number of possible transitions from the current state, we will have those many number of functions in this class that would in turn serve as the hyperlinked [8] resource functions. These would have to be coded in this class.

Resource specific information in this file include,

- **Name of the file**

  The name of the file will have the name of the resource as a prefix to the *RestController.java* to associate it with the resource. Naming the file such will also make it easier to locate.

- **Name of the class**

  The name of the class will be same as that of the file to make things simpler. The name of the resource will be used as a prefix to the *RestController*. This name will be later used in the *ResourceAssembler* class, explained later in this section.

- **Name of the persistence repository**

  A repository in the Spring [7] world is an interface that has functionalities to perform persistence operations on the database. It is annotated with *@Repository* [7] and would also extend the *JPARepository* [7]. The *JPARepository* will have two

attributes, first will be the entity [7] class and the other will be the datatype of the identity field, which is our case has been standardised to *Long*. We will use this as the datatype of identity for both resource and entity.

The *RestController* will have many functions that will need to perform persistence operations. Injecting a repository will enable the *RestController* to have functionality to perform such operations. So declaration of the repository has an advantage. Injecting is done with the use of *@Autowired* [7] annotation before the declaration of the repository.

- **HTTP method types**

  For every function we have to specify what kind of HTTP [1] method type would trigger the function. This is achieved by declaring an annotation *@RequestMapping* [7] with attribute *method*. The value of *method* is an enumeration called *RequestMethod*. It can have the enumeration types GET, POST, PUT, DELETE, PATCH, OPTIONS etc. [28]

  This contributes to the combination of HTTP [1] method type along with the URI. Every function that represents a resource must have a unique combination of HTTP method type and URI [1].

  We have to make sure that we represent all the functions declared in the rest controller with a HTTP method type, since there would be no default method type.

- **URI mapping**

  Every function that is declared in the rest controller has to be mapped to a URI [6]. URI mapping can be achieved by declaring an annotation *@RequestMapping* [7] with attribute *value*. The *value* can be a String value with '/' used to mark beginning of a relative path or relationship.

  In Spring [7]'s rest controller, a hierarchical approach towards mapping is followed. The first level of hierarchy is the class and then the function. So the class itself can be annotated with *@RequestMapping* [7]. If done so, then the functions within the class will all be prefixed with the mapping of the class. If not, then the functions will follow its own mapping.

  In our scenario we see that a *PurchaseOrderResource* will have many functions within its rest controller. Some of the function will have a separate mapping, like the ones that represent state change etc. But they all belong to the resource *PurchaseOrder*, so hence we can map the class itself as 'purchaseorder' so that everything that follows will have a relationship with '/purchaseorder'.

  This makes it easier to locate a resource on the web. Also it would represent the container class of this resource. After a resource is created it would have a unique Identifier, which would be long, and a new relation would be made relative to the current URI [1] which would define the URI for the particular resource, an individual *PurchaseOrder*. The relation would look like '/purchaseorder/<identity>'.

- **Functions for state transitions**

  Every state transition that we define will have a corresponding method in the controller that will perform the function of state change. This function will be triggered by a HTTP [1] call based on the trigger and method type. The transition functions will only be available in the controllers of the resources that have states. The resources that do not have any states will not need to have these functions since there would be no transitions.

  Transitions functions are mostly same as regular functions in declaration, i.e. they have an annotation *@RequestMapping* with attributes *method* corresponding to spring's RequestMethod [6] enumeration and *value* corresponding to the string relative path for the URI [1].

  Apart from that the function contains some internal logic which is basically the change of state from one to another with the use of the autowired spring repository [7] interface.

- **CRUD functions**

  CRUD basically stands for Create, Read, Update and Destroy. These are the most basic of the persistence functions. The names are pretty much straight forward, giving us an understanding of the functions. In essence every CRUD function has a corresponding HTTP [1] request function type. Create can be achieved by a POST [28] request, Read can be achieved by a GET [28] request, Update can be achieved by a PUT or PATCH [28] request and finally Destroy can be achieved by a DELETE [28] request.

  These are merely conventions but play a vital role in modelling of robust and understandable REST [1] application programming interfaces. On the other hand since they are merely conventions, practice has shown a variety of usages and one which does not break the code but would simply make it hard to understand. In order to follow a standard we will follow the correspondence mentioned above.

  These functions, similar to other functions in the controller, are annotated with *@RequestMapping* [6] with method and value attributes whose values correspond to spring's RequestMethod [7] enumeration and path relative respectively. Some functions are made to map on the container while some on individual resource. In case it is mapped on the container then the *value* attribute of mapping will hold a relation without the unique identifier and simply imply the container. Whereas if the function is mapped on an individual resource then the unique identifier of the resource must be known while making the request, hence by simple convention we put it as a path variable. The path variable should also be defined within the functions signature as a parameter with it's type, which in case of identity will be Long along with an annotation *@pathvariable* [7].

  According to spring's conventions, every resource has five basic CRUD functions.

  - GET on the container

    This will return a list of all resources within the container.

    GET /purchaseorder

- GET on the individual resource with identity as path variable.

  This will return the individual resource identified.

  GET /purchaseorder/<identity>

- POST on the container

  This will create a new resource in the container.

  POST /purchaseorder

- PUT or PATCH on an individual resource with identity as path variable

  This will update or change state of the individual resource identified.

  PUT /purchaseorder/<identity>

  with PurchaseOrderResource as RequestBody

  Or

  PATCH /purchaseorder/<identity>

  With only a some fields of PurchaseOrderResource that need be altered as request body.

- DELETE on an individual resource

  This will delete the individual resource identified.

  DELETE /purchaseorder/<identity>

4.1.1.6 PurchaseOrderResourceAssembler.java

This is a class that would extend *ResourceAssemblerSupport* of Spring [6]. The *ResourceAssemblerSupport* class provides a bridge between an Entity and it's resource. Once again this would mean that this class is not applicable to 'external resources', explained in section 4.3, since they would not have any entity to represent in their domain model.

So for all 'internal resources' this class would provide an easy conversion of the domain model information to it's resource model counterpart. In classic sense this class would assemble a resource, and make ready for transfer.

This class would be responsible for adding hyperlinks [6] to a resource with the use of many convenient functions. Hence the behavioural aspects [2] of a resource will play a vital role in writing this class. Based on the current state of a resource the hyperlinks [6] to be added will vary.

For e.g. in the Plant hire scenario [13] above, PurchaseOrderResource could transit from state *PENDING_CONFIRMATION* to either *OPEN* or to *REJECTED*, hence at state *PENDING_CONFIRMATION* it will have two hyperlinks, but once it transits to *CLOSED* then there are no more transits available hence in this state it will have no hyperlinks [6].

The *toResource()* function of this class will be extensively used in the rest controller as it would assemble the resource for delivery in almost all the functions with HTTP method type as GET [28].

4.1.1.7 PurchaseOrderStatus.java

This would be an enumeration of all the different states that a particular resource can be in during the application's lifecycle. This will be simple and only the information about the states would suffice during coding.

This would only be a part of 'internal resources', see section 4.3. This will be an attribute in the entity class of this resource.

4.1.1.8 PurchaseOrderNotFoundException.java

Not the most interesting ones, but this exception would be thrown when a particular resource is not found by the controller when it's GET [28] function is invoked.

4.2 Syntax

In this section we will address the syntax of our Domain specific language [9]. This section differs from previous section. In the previous section we addressed the actual syntax in Java to achieve a REST [1] application. In this section we will address the DSL that we will build to incorporate all information needed, the ones we gathered from the research from the previous section, in order to build a REST application with spring HATEOAS [6]. XText [14] provides a nice platform to create rules and logic for preparing a DSL. Making use of xtext's features we would build an editor designed to take all inputs.

The motive is to collect all Structural and Behavioural [2] aspects of the model from the user with minimum syntactic complexity. Considering this requirement we need to come up with a simple, understandable syntax for the user.

4.2.1 Terminology

For the language that is needed, this section will enlighten some special terms you need to know. These terms have been used to describe various elements in the syntax of this language. In order to understand the examples used while explaining these terms, you would need to read the Plant hire scenario [13] in section 4.1.

- ***Package***

A spring project like other Java [26] based frameworks relies on packaging the classes in proper order. The package generally would hold the 'base package' for the project. E.g. *ee.ut.rentit* etc. A package would be the first information to be declared in the DSL [9]. A package could be declared by using the keyword *package* followed by the package string, such as the following,

package ee.ut.rentit

- ***Database Configuration***

Every project requires database configuration to be able to perform persistence operations. Spring HATEOAS [6] is no exception, configuration of database must be done at the project level. In our DSL this would be the second declaration after the *package* declaration. Database configuration will consist of the following inputs,

- Database

  Will choose the database name that would be used by the application. Declared using keyword *database.* Would be a string declaration hence in single quotes.

  database 'postgres'

- username

  Will provide username. Declared using the keyword *username.* Provided in single quotes.

  username 'postgresuser'

- password

  Provides password for the access. Declared using keyword *password.* Written in single quotes as string.

  password 'psqlpassword'
- host

  Provides hostname. Declared using keyword *host.* Provided in single quotes.

  host 'localhost'
- port

  A numerical value that would provide the port for connection. Declared using keyword *port.*

  port 5432

  This all information would be provided with the *dbconf* declaration. The above configurations would be embraced in curly braces after the *dbconf* keyword.
  E.g.

```
dbconf {
        database 'postgres'
        username 'postgresuser'
        password 'letmein'
        host 'localhost'
        port 5432
}
```

- ***State***

A State is one particular user-defined and named state [6] of a particular resource. It is declared within the embracement of the *states* keyword and simply noted by it's name which is a string. If there are more than one states then they are separated by a comma.

Preferably this string could be one which follows the Java naming conventions for constants which is a capital snake-case fashioned constant declaration, but the user has the freedom to name it as preferred.

E.g. from the scenario [13],

PENDING_CONFIRMATION

- ***States***

States is a collection of multiple user defined State. This collection can qualify as an enumeration. We need this information because we need to know the different states that a particular resource can have.

It is defined with the keyword *states* and the information is wrapped around within two curly braces and separated by a comma.

E.g. from the scenario [13],

states {

PENDING_CONFIRMATION,

OPEN,

ACCEPTED

}

- ***Transition***

A transition as the name suggests is a transition between two states. A transition will have numerous sub-entries to be filled by the user.

  - Trigger combination

    A transition will have a trigger in the form of a combination of a *HTTP method type* [28] and a relation to the current resource. A relationship or *rel* is a term that spring [6] uses to define a self-referencing hyperlink. This means that a particular resource can have a link to itself but after performing minor change to itself. In this case the change would be change in state. A *rel* will make sure that this transition will create a resource for the action that this transition performs along with its URI. A combination of *HTTP method type* and a *rel* will make every transition in the resource a unique one.

    Input for *HTTP method type* could be one of the four compatible method types GET, POST, PUT, DELETE [5] with the keyword *with*. Input for *rel* will be a single quoted string with the keyword *on*.

  - From state

    This one will define the state from which the transition would take place. This means that when this transition takes place, the current state of the resource must be what is mentioned here. Input can be one of the defined states with the keyword *from*.

  - To state

    This one will define the new state of the resource. This means that after the transition takes place the state of the resource would change to this one. Input can be one of the defined states with the keyword *to*.

  - Controller function

Every transition will have some effects on the resource. In the simplest of situations it will be the change in state. For this to happen, the logic has to be coded in the rest controller. The controller function is usually referenced while creating links hence there is a need to know the name of the controller function. In reality this should not be needed as we can create a name of our own, but this functionality will allow the user to give a name in accordance with the Java naming conventions for functions.

Input will be in form of a single quoted string with the keyword *using*. The Syntax for transition will be as follows,

transition <*fromState*> *to* <*toState*> *using* <*controllerFunction*> *with* <*methodType*> *on* <*rel*>

E.g. from the scenario [13],

transition OPEN to ACCEPTED using 'acceptPO' with POST on 'acceptance'

Here OPEN is a state and ACCEPTED is another state. When a resource is at state OPEN, it will have a relationship which would transit the resource to another state which is ACCEPTED. This hyperlink accepts a POST request on the current resource and the logic for changing the state will be defined in 'acceptPO' function of the controller.

4.2.1.1 Internal resource

An Internal resource is one which is specific to the domain of the application. In order words this resource originates at the application and has an entity representation in the domain model.

These resources have an entry in the database. These resources are the ones which have been created from an entity in the domain model. Every entity will have a collection in the database. Every collection can be converted into a container for the resource type.

From the Plant hire scenario [13] above, the application domain has a resource called PurchaseOrder. This resource is a part of the application's domain model and will have an entry in the database. A collection of this entity can be converted into a list of PurchaseOrderResource which will serve as a container, and a specific PurchaseOrderResource can be found by its unique identifier. We can make sure of this by putting the identity of this by creating a URI from the container. And further creating an URI for each resource with the identifier on the path or the URI as a path variable [6]. With this we can make sure that every PurchaseOrderResource will have an URI which will hold its identity and have a unique URI for itself. Also the container for PurchaseOrderResource will have a unique identity for itself.

An internal resource can be created using the keyword *internal* followed by other information which is enclosed in curly braces.

- Datatypes

  An internal resource will first have some data types. These datatypes are the same as JVM provided datatypes such as String, Long etc. These will follow a simple declaration syntax.

<javaType> <name>

Eg,

java.lang.String myName

- Internal Datatypes

  An internal resource itself can serve as a datatype for another internal resource. We come across several scenarios where a resource comprises of other resources. Take an example from our plant hire scenario above, a PurchaseOrder will have a Plant.

  In this case we need two things. One is that we need the resource which is to be placed as a datatype in the internal resource, to be defined before we define the current one. Second, we need to provide cardinality information. Along with this we also need to provide a name to this datatype as an unquoted string. We have to prefix the datatype with a cardinality one from the following: OneToOne, OneToMany, ManyToMany, ManyToOne.

  The syntax for an Internal Datatype is as follows,

  <cardinality> <internalDatatype> <name>

  E.g. from the scenario,

  OneToOne PurchaseOrder purchaseOrder

  The syntax for an Internal consists of the keyword internal followed by first the datatypes then the internal datatypes embraced in curly braces.

  internal <name> {

  String datatype1

  int datatype2

  OneToOne <InternalDatatype> <name>

  …

  }

4.2.1.2 External Resource

An External resource is one that does not originate at the application domain but either in the business logic of the application or in domain of another application that would interact with the application. If originating in another application, this can come to the application as an input along with the request from the originating application. This case is considered only for situations when predevelopment knowledge of such interaction will be known, like in case of the Plant hire scenario [13], where the knowledge of *RentIt* and *BuildIt* interaction is known. The other situation is when the resource originates at the application but does not have a representation in the domain. This resource is important for the application because it serves as an information carrier that the application uses to fulfil its business logic.

Since this resource originates from another application or it does not necessarily have a place in the application's domain model, hence this resource would not have an entity

entry in the database. And hence there would be no need for creating entity specific files for such a resource.

In the Plant hire scenario [13] above, we see one such resource called *PlantHireRequestResource*. This resource does not originate from the application but since it has certain information such as start date and end date etc. which are could be beneficial for the creation of a *PurchaseOrder* and its resource, there is an option to create it.

- Datatypes

  Same as an Internal resource, an external resource will also have some datatypes provided by the JVM [12] such as int, float, String etc. These will serve as the properties of this class and will determine the properties that the resource carries. The syntax is same as that of an Internal resource's datatype.

  <javaType> <name>

  E.g.

  java.lang.String myName

  The syntax for an External consists of the keyword external followed by the datatypes embraced in curly braces separated by a line.

  external <name> {

  String datatype1

  Float datatype2

  ...

  }

5 Code generation

5.1 Parsing Input

In the earlier sections we learned about what information is required to code the language in our chosen scope and framework. Also we came up with a Meta-model i.e. the syntax to create a Domain specific language (DSL) [9] to take it as input from the user. This section will deal with creating the end code i.e. a spring REST application [6] from the DSL inputs we take from the user.

To achieve this we would be making use of Xtext [14] framework.

5.1.1 About Xtext

Xtext is framework which is based on eclipse [15], which provides useful set of rules and definitions that provide a good base for creating one's own grammar. With Xtext [14] one can accomplish parsing, linking and compiling. That along with a possibility to export the product out as a plugin to the two most popular Integrated Development Environments for Java: intellij idea [29] and eclipse [15], makes it an ideal choice for this part of the research.

5.1.2 Requirements

To begin with we need an IDE [30] to build our Xtext [14] DSL [9]. During the course of research done for this paper, the IDE used for developing the DSL was Eclipse Mars.1 Release (4.5.1) which has an out of the box support for Xtext. But in general to be able to use Xtext one may do any of the following,

- Install a fresh installation of eclipse with Xtext support available on eclipse downloads section [31] on the website.
- Or Install the Xtext plugin [32] and Xtend plugin [33] available in the plugin section on eclipse' website onto any eclipse based integrated development environment, e.g. standard Eclipse release, Spring Tool Suite etc.
- Or Install the Xtext plugin [34] available in the plugin section on eclipse' website onto intellij idea.

5.1.3 Getting started

Once we have our IDE [30] we can begin with making the DSL [9]. Create a new Xtext [14] project by new > project > xtext project.

Select a project name. This will correspond to your project. The field language name corresponds to the name of your language that you create. In the case of this project the name selected was restgen.rgdsl and prefixed with the domain of the University of Tartu. For the sake of simplicity let us call this as our base package. As for the name of the language, the selections was RgDsl. For the sake of simplicity let us call this Language name.

Project name: *ee.ut.restgen.rgdsl*

Language name: *ee.ut.restgen.rgdsl.RgDsl*

Extension: *.rg*

You may opt to select SDK support, UI support. In this paper we will only address the DSL [9]. Once you have created your project you will notice your project package. In the *src* folder you will notice the base package, or in other words the package that you selected while creating the project, along with some other packages as well. In the base package you will notice a file named after your language and with the extension you opted to use. In our case, *RgDsl.rg* was the name.

This file would be the main grammar file where all the rules will reside. In this file one can create the parser with the syntax discussed in the previous section. After much research and work the writing of the DSL [9] was complete. In order to do so, some basics were needed to be understood.

- Parser-Rules

  The first two lines are the package and generator instance respectively. The main logic behind the DSL [9] starts after this. Every declaration in xtext [14] takes place in terms of *ParserRules* [35]. These are basic set of rules that define the every elements of the language. In other words every element in the language is a *rule*. A rule may consist of other rules. The first and foremost rule is the main parser-rule. Everything within the language has contained by this rule.

- Keywords

  Everything that qualifies as keywords within the language would come under single quotes. It may be placed within a rule or before or after a rule. There are some reserved keywords that xtext has placed for special purposes. Some of the encountered keywords are below, [35]

Keywords	Meaning
‘.’	Placing a dot between IDs can manipulate the qualified name
ID	Unique identifier of a rule marked by the assigned name
name	A variable serves as the name of the rule and assigned with a String
enum	Enumeration
terminal	Any form of ordered Rule like Number system etc.

Table 4. Native keywords of Xtext

- Operators

  Xtext [14] provides a variety of operators of which some had been used. The operators used in the DSL [9] for this research are, [35]

Operator	Meaning
+=	Used to declare a collection of a Rule
*	Used to multiple occurrences including none
=	Used for declaring a variable to a Rule
‘ ‘	Used for declaring keywords strings
..	Used for declaring a Range (numbers, alphabets etc.)
|	Logical OR operator. E.g. *Rule1* | *Rule2* etc.
[ ]	Used to reference a rule defined
:	Marks the beginning of a rule definition
;	Marks the end of a Rule definition

Table 5. Operators of Xtext

After following the conventions of xtext [35], the DSL incorporating the required syntax has been prepared and a copy of it is available in the appendix as *RgDsl.rg*

Once the DSL has been created we can generate xtext artefacts [35]. This would allow us to treat the language we just created as an incorporation in an editor. By default eclipse editor has an option to include xtext artefacts if they are available by means of applying the xtext nature to the parser logic.

To generate xtext artefacts, in eclipse select the run as menu of the language file and choose the option Generate Xtext Artefacts. [36] This will create the required files and once done you will notice a .mwe2 file in your base package. This file consists of all project level information. In order to place the generated files in proper folders, we realised the need to manipulate the workflow section of this file. As well as there was a need to create a class that would extend *IOutputConfigurationProvider* and override the default output configuration. This solution was found on the internet as open information. [37] By doing this one would enable the files generated to be placed in a proper folder structure, which in our case is as per a spring boot project. The file was named *RestGenOutputConfiguration.java*. Refer to the following subsection for the code generation mechanism and to the appendix for the above mentioned file.

5.2 Generating code from Input

In the above subsection we dealt with the challenge of creating an editor with a parser that would enable use of the syntax we considered. So far we have managed to get input from the user in the format that we wanted. Now comes another challenge, the phase of generating output code from the input.

There will be a file generated which generating the artefacts, having extension *.mwe2*, in the base package. This file as mentioned before this file holds all the project level

information and will help in generating the end code as well. In eclipse select this file, and select the MWE2 Workflow option from it's run as menu. This will create all the generators for the DSL. [38]

Alongside the base package you will notice another package suffixed *.generator*. In this package there would be an *.xtend* file. This is the main generator file and all the logic behind end code generation would reside here. Similar to the DSL writing in the previous section, writing logic to this file also needed some understanding of common basics. After research and understanding of this *.xtend* file, the generator mechanism was written. Refer to *RgDslGenerator.xtend* in the appendix for the code. [38]

The file will contain a class named after the language name chosen, and this class will implement *IGenerator* interface of xtext. [39] The *IGenerator* interface has a function *doGenerate* which has to be overridden. This function will be invoked during generation process. There are two input parameters to this function. First, the instance of the *Resource* class which is an xtend representation class for the *.rg* model that we would write in the editor. Second, an instance of *IFileSystemAccess* class of xtext [39]. This instance is created by xtext automatically during the generation process based on the output configuration we setup as explained in the previous subsection.

With the use of the *IFileSystemAccess* (FSA) instance we can create files with specified contents. The logic behind the content of each file was based on the research done in the section 4 i.e. the file contents required by as per a fully working Spring HATEOAS [25] project following naming conventions. We already know what files are needed and their names. We could use the FSA instance to create a these files and give them the content string with the use of it's *generateFile* function [39]. This function would take as input parameters the name of the file including the source directory relative to the project's main directory. We know the name of the file but as for the source directory, we would get that information from the *fullyQualifiedName* property [38]. This holds the name of the element along with the base package we specified. We could extract the base package and then convert it into directory type '/' delimited string which would in turn give us the file name with it's proper source directory. The second input parameter is the content of the file. For the content one can simply provide a string e.g. 'Hello world!' etc. In our case the contents would most definitely be more than 'Hello world' and involving some amount of logical derivation, hence it was easier to use a separate function for each file to be derived which would take care of the logic behind writing specific lines.

5.2.1 Contents

The contents of the each *.java* file could be divided for each into three sections. First, the package of the file. Second, the imports area which will hold the imports from other packages within the application. Third, the body part which would contain the class declaration and all the Spring annotations etc. As mention above we would use the facility of function definitions within our generator to provide the contents for each files.

A function declaration can be declared by use of the xtext keyword *def* followed by the input parameters to the function encompassed in round braces i.e. *'('* and *')'*. The body of the function serves as a string building area. The body is encompassed between two separate declarations of three consecutive single quote characters i.e. *'''*. What lies between these two declared *'''* and *'''* is what will be written into the file. [38]

Some files that are needed to be generated are not resource specific rather application specific files. They will not hold any resource specific contents, hence will be created once. For the files that are resource specific, we will have to create them for all the resources

declared in the *.rg* file. In the *doGenerate* [39]function we make use of the *generateFile* [39] function of the FSA instance to create a file. For files that are resource specific, it was achieved by looping through the instance of Resource, which is the first parameter to the *doGenerate* function and is an iterable that contains all the elements of the DSL, which in our case would include *External* and *Internal* resources. Further we then called the function definitions we created for internals and externals respectively by making a simple check. The declaration for the creation of the non-resource specific files were made outside the loop and would be created only once.

An xtend [38] function also allows special operations within these boundaries. These operations can be performed by use of logic within logical blocks. Anything that is written outside any logical block will not be treated to perform dynamic content operations, but will be considered merely as strings. [38] During the course of this research there were encounters with a few of them and they have been listed below. [35]

Keywords	Meaning
«	Beginning of dynamic content logic block
»	End of dynamic content logic block
<IF condition>	Begin if block with condition
<ENDIF>	End of if block
<FOR variable: collection>	Begin of For block with input collection and an iteration variable
<ENDFOR>	End of For block
val	A value object. Needs to be declared within a dynamic content block.

Table 6. Xtend keywords and Operators

For writing the imports we needed to declare an import manager instance. Xtext provides a class called *ImportManager* which holds information of all JVM based datatypes declared within a rule while writing the *.rg* file. This solution was found on the internet as an open information. [40] Looping through the import manager instance helped us create a dynamic logic to write the imports for each files. In many files some non-JVM imports were needed but almost all of them were known to us since they were all part the same project and spring imports. Hence it could be achieved with use of for loops. With use of Import-Manager and for loop we managed to write the logic to declare imports.

For the body, some of the files needed dynamic contents but it was manageable with use of above mentioned dynamic content blocks. When the files are enerated the yare placed in their proper source directory because of the configuration placed in *RestGenOutputConfiguration.java.* This will override the default configuration. The files generated can be overwritten at any time. The generation merely creates an initial version for the user and the user can later edit the files manually or make changes to the *.rg* file again to make new changes.

5.2.2 Resource specific files

These files will be generated for all the *internal* resources declared in the DSL [9]. The following files would be generated for a resource called *PurchaseOrder* from the Plant Hire Scenario [13]. With the use of *fullyQualifiedName* [39] function of xtext [14], it is possible to extract the package information of *Internal e.* The last segment of the *fullyQualifiedName* is the name of the internal itself, hence we need to skip that to get the proper base package. All these files contain resource specific information in a certain places. The input resource parameter for the content generation functions of all the files below example would be let's say *Internal e*.

- PurchaseOrderResource.java

  - Package

    These range of files will be placed in *.rest* after the base package.

    package «e.fullyQualifiedName.skipLast(1)».rest;

  - The *@XMLRootElement* annotation [7]

    This will contain the name of the resource in lower cases as a string value to the attribute *name.* The name property of *internal e* will give the name of the internal resource.
    Usage: -

    name="«e.name.toLowerCase»"

  - The class declaration

    This will contain the name of the resource with first letter capitalised suffixed with 'Resource' to maintain naming conventions. It will also extend the *ResourceSupport* class that would incorporate some resource generation features.
    Usage: -

    public class «e.name»Resource

  - JVM declaration

    For the JVM declaration we would loop through the *datatypes* property of *Internal e*, which as per our DSL [9] is a list of all JVM [12] datatypes declared in that *internal.* Along with that we must also declare getters and setters for the same.
    Usage: -

```
«FOR f:e.datatypes»
    «manager.serialize(f.dataType.type)» «f.name»;

    public void set«f.name.toFirstUpper»(«f.dataType.simpleName» «f.name») {
        this.«f.name» =  «f.name»;
    }

    public «f.dataType.simpleName» get«f.name.toFirstUpper»() {
                return «f.name»;
    }

«ENDFOR»
```

  - Internal resource declarations

For the internal resource declarations we have followed the same steps and made use of a for loop [35]. Looping through property internals of an internal will give us individual internal resource datatype declarations.
Usage: -

```
«FOR i:e.internals»
    «i.internal.name»Resource «i.internal.name.toFirstLower»Resource;

    public void set«i.internal.name.toFirstUpper»Resource(«i.internal.name»Resource «i.name»Resource) {
        this.«i.name»Resource =  «i.name»Resource;
    }

    public «i.internal.name»Resource get«i.name.toFirstUpper»Resource() {
        return «i.name»Resource;
    }
«ENDFOR»
```

- PurchaseOrder.java

  The resource specific information to be placed in this file is explained with the use of *Internal e* as the parameter.

  - Package

    These range of files will be placed in the package .*models* after the base package.

    package «e.fullyQualifiedName.skipLast(1)».models;

  - Name of the class

    The name of the class will have the name of the resource with first letter capitalised.

  Usage: -

  public class «e.name»

  - JVM declaration

    Like the earlier file this will be similar. We need to declare all JVM datatypes declared in the DSL [9] along with their getters and setters.

  Usage: -

```
«FOR f:e.datatypes»
    «manager.serialize(f.dataType.type)» «f.name»;

    public void set«f.name.toFirstUpper»(«f.dataType.simpleName» «f.name») {
        this.«f.name» =  «f.name»;
    }

    public «f.dataType.simpleName» get«f.name.toFirstUpper»() {
                return «f.name»;
    }

«ENDFOR»
```

  - Other entity declarations

    Unlike the previous file this one will have the declarations of other Entities. The cardinality will also play a role here. The use of property *cardinality* will play a role. In a spring application cardinality is denoted by a declaring one of spring supported cardinality annotations. [7]

  Usage: -

```
«FOR i:e.internals»
    @«i.cardinality»
    «i.internal.name» «i.internal.name.toFirstLower»;

    public void set«i.internal.name.toFirstUpper»(«i.internal.name» «i.name») {
        this.«i.name» =  «i.name»;
    }

    public «i.internal.name» get«i.name.toFirstUpper»() {
        return «i.name»;
    }
«ENDFOR»
```

- PurchaseOrderRepository.java

  The resource specific information to be placed in this file is explained with the use of *Internal e* as the parameter.

  - Package

    These range of files will be placed in the package *.repositories* after the base package.

    package «e.fullyQualifiedName.skipLast(1)».repositories;

  - Interface declaration

    The Interface declaration will contain the name of the resource

    Usage: -

    public interface «e.name»Repository extends JpaRepository<«e.name», Long>

- PurchaseOrderRestController.java

  The resource specific information to be placed in this file is explained with the use of Internal e as the parameter.

  - Package

    These range of files will be placed in the package *.rest.controllers* after the base package.

    package «e.fullyQualifiedName.skipLast(1)».rest.controllers;

  - Class declaration

    The class declaration will contain the name of the resource.

    Usage: -

    public class «e.name»RestController

  - Repository injection declaration

    In usual practice all persistence operations in spring application's controllers are done by the repository injects. The repository injections will include the repository interface declaration of the controller class in question as well as the repositories of all the internal datatypes of that resource.

    E.g. in PuchareOrderRestController we would have the PlantRepository as well as PurchaseOrderRepository.

Usage: -

```
@Autowired
   «e.name»Repository «e.name.toFirstLower»Repository;
«FOR internal: e.internals»
      «internal.name»ResourceAssembler «internal.name.toFirstLower»ResourceAssembler =
        new «internal.name»ResourceAssembler();
«ENDFOR»
```

- Class request mapping

  The class itself will be mapped with the name of the resource. This would allow proper URL structuring by making sure that every function within the class will be mapped in relation with the main mapping i.e. the name of the resource. [7]

  E.g.

  GET /purchaseorder/<identity>
  POST /purchaseorder
  GET /purchaseorder/<identity>/plant

  All of these are part of the same controller and they have one thing in common, the leading relation on the URL mapping i.e. 'purchaseorder'. This is achieved by mapping the class itself.

  Mapping the class is achieved by,

  @RequestMapping(value = "«e.name.toLowerCase»")

- Uses of instances of the resource

  In a lot of cases there will be known usages of the instances of resource related Classes within the controller. These could include the resource assembler or resource class itself or the repository. By use of naming conventions one can properly judge where the name of the resource would appear in such usages and based on this we managed to place the name of the resource in proper places so as to achieve the outcome.

  E.g. The following is the generic function to create a GET function for a resource identified by it's identity,

```
@RequestMapping(method = GET, value = "{id}")
   public «e.name»Resource get«e.name»(@PathVariable Long id) {
      «e.name» «e.name.toFirstLower» = «e.name.toFirstLower»Repository.findOne(id);
      return «e.name.toFirstLower»ResourceAssembler.toResource(«e.name.toFirstLower»);
   }
```

  Take a look at the different places where we have used *«e.name» or «e.name.toLowerCase»* to achieve the objective.

- Controller functions

  We know a few functions that a Rest controller will have for every resource. But many of them depend on the *states* and *transitions* defined in for that resource in the DSL. This part incorporates the behavioural aspects of the resource, as it uses the input and it's state transition information to create functions in the controller that are mapped accordingly. To do this we needed to loop through the property transitions of *Internal e,* which according to the DSL we studied earlier, is a collection of all the transitions defined for that internal.

Usage: -

```
«FOR transition: e.transtions»
    @RequestMapping(method = «transition.methodType», value = "{id}/«transition.rel»")
  public «e.name»Resource «transition.controllerMethod.toFirstLower»(Long id) {
    «e.name» «e.name.toFirstLower» = «e.name.toFirstLower»Repository.findOne(id);
    «e.name.toFirstLower».set«e.name»Status(«transition.toState.name»);
    «e.name.toFirstLower»Repository.saveAndFlush(«e.name.toFirstLower»);
    return «e.name.toFirstLower»ResourceAssembler.toResource(«e.name.toFirstLower»);
  }
«ENDFOR»
```

- PurchaseOrderResourceAssembler.java

  The resource specific information to be placed in this file is explained with the use of Internal e as the parameter.

  - Package

    These range of files will be placed in the package *.rest.utils* after the base package.

    package «e.fullyQualifiedName.skipLast(1)».rest.utils;

  - Class declaration and constructor

    The class declaration will contain the name of the resource. And the constructor of this class will contain the name of the resource in two places, one for the name of the rest controller class and the other for the name of the resource class.

Usage: -

```
public class «e.name»ResourceAssembler extends ResourceAssemblerSupport<«e.name», «e.name»Resource>{

  public «e.name»ResourceAssembler() {
    super(«e.name»RestController.class, «e.name»Resource.class);
  }
```

  - *toResource* function

    The toResource function is a major function of this class. This function plays a major role in the behavioural aspects of a HATEOAS project. The main objective of the function is to assemble the resource with all it's properties and hyperlinks. Everytime a resource is created it will have a certain hyperlinks which are merely performed by the *add* function of the resource class. The add function comes from *add* function of *ResourceSupport* class that we defined. In order to known the hyperlinks we need to know the possibilities based on the current state of the resource and the transitions defined. A hyperlink will be a URL link embedded within the resource. For instance a state change can be invoked by making a POST request to a particular URL defined as a link within the resource. This means that the links will be different as the resource transits from one state to another. This information will be available in the property transitions of *Internal e,* in our example. In our end code we would make use of a switch case in Java to achieve the link adding process.

    So for the generator we made use of a for loop [35] again to loop through the transitions property.

Usage: -

```
switch («e.name.toFirstLower».get«e.name»Status()) {
        «FOR state: e.states»
        case «state.name»:
        «FOR transition: e.transtions»
        «IF state == transition.fromState»
                «e.name.toFirstLower»Resource.add(
                new ExtendedLink(linkTo(methodOn(«e.name»RestController.class)
                        .«transition.controllerMethod.toFirstLower»(«e.name.toFirstLower».getId()))
                        .toString(), "«transition.rel»", "«transition.methodType»"));
        «ENDIF»
        «ENDFOR»
                break;
        «ENDFOR»
        default:
           break;
     }
```

- PurchaseOrderNotFoundException.java

  The resource specific information to be placed in this file is explained with the use of Internal e as the parameter.

  - Package

    These range of files will be placed in the package *.rest.exceptions* after the base package.

    package «e.fullyQualifiedName.skipLast(1)».rest.exceptions;

  - Class declaration and constructor

    For the class declaration and the constructor, simply the name of the resource is enough.

    Usage: -

```
public class «e.name»NotFoundException extends Exception {
   private static final long serialVersionUID = 1L;

   public «e.name»NotFoundException(Long id) {
      super(String.format("«e.name» not found! («e.name» id: %d)", id));
   }
}
```

5.2.3 Spring application files

The spring HATEOAS application requires some classes to be written. These files are not resource specific hence will not involve *Internal* or *External* resource information for their generation. One of these files will require database configuration information from our input DSL. Since these files are common for the entire application, they will be generated only once hence we do not need to loop through any elements of our DSL but one which is the main element or main rule of our DSL since that is where all the project level information will lie when writing into the *.rg* file. This is unlike the files in the above subsection which depended on *Internal* and *External*. For these files only project level information would be needed, more specifically the database configuration which is available in the property *databaseConfiguration* of the main rule (ResourceModel) as per our DSL. Along with that we need the name of the project which is available as the last segment of

the base package. The base package is available with the use of function *fullyQualifiedName*, which is an *xtext* function that holds the package information for every element in the DSL. This function can be applied to every element in the *.rg* file. Let us assume our project is named 'Rentit' and the package that we set is 'ee.ut.rentit', the following would be the files generated that follow this specification,

- RentitApplication.java

  This file would be the main file of the application. This file is a runnable file and this is the file that starts up a Spring application. The information required for this file is only the name of the project. We get this information from the base package by extracting the last segment from that. The input to our content generation function for this would be the base package which is available with the use of *fullyQualifiedName* function.

  Package: -
  This file will be placed in the base package itself.

  package «basePackage.toString»;

  Content generation: -

```
def createApplication(QualifiedName basePackage) '''
package «basePackage.toString»;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class «basePackage.lastSegment.toFirstUpper»Application {

   public static void main(String[] args) {
      SpringApplication.run(«basePackage.lastSegment.toFirstUpper»Application.class, args);
   }
}

'''
```

- RentitDatabaseConfiguration.java

  This file will configure the database for the project. As per the scope of this research only Postgres has been incorporated. The information required for this file is available in the *databaseConfiguration* property of the main rule of our DSL. The parameter to the content generation function would be the main rule in the DSL, in our case *ResourceModel rm*.

  Package: -
  This file will be placed in the base package itself.

  package «rm.fullyQualifiedName.toString»;

  Content generation: -

String username = "«rm.databaseConfiguration.username»";
String password = "«rm.databaseConfiguration.password»";
String url = "jdbc:postgresql:// «rm.databaseConfiguration.host»:«rm.database-Configuration.port»/«rm.fullyQualifiedName.lastSegment»-test";

The following files only need base package information for generation. They will both be placed in the package *.rest.utils* after the base package.

package «rm.fullyQualifiedName.toString».rest.utils;

- ExtendedLink.java [13]
- ResourceSupport.java [13]

6 Installation

To install the application you must have installed an eclipse based IDE or intellij idea. The project is setup as an plugin on a github [41] page. The address to it is the following, [41]

http://*vishalkirandesai.github.io*

6.1.1 Eclipse based IDE

1. Open up the eclipse IDE.
2. Open *Help > Install new software…*
3. Select *Add*.
4. Write any name in the *Name* box.
5. Write the address above to the *Location* box.
6. Tick the plugin from the list and select Finish.

6.1.2 Creating a project

If you have installed STS then you can create a Spring boot app [25] from the IDE itself. If you have another eclipse based IDE then you would have to create an initialiser from the spring initializr [42] website. *https://start.spring.io/*

While creating a new Spring starter Project, either STS or from initializr, select the following components to be added to your Project,

- Web
- Web services
- HATEOAS
- Jersey (JAX-RS)
- Rest repositories
- PostgreSql

Once created some more dependencies need to be added before we can begin. In the pom.xml file in your Project, in the dependencies section, add the following dependencies,

- Codehaus Jackson [44]

```
<dependency>
   <groupId>org.codehaus.jackson</groupId>
   <artifactId>jackson-mapper-asl</artifactId>
   <version>1.9.13</version>
</dependency>
```

- Commons-dbcp [44]

```
<dependency>
   <groupId>commons-dbcp</groupId>
   <artifactId>commons-dbcp</artifactId>
   <version>1.4</version>
</dependency>
```

Once the Project builds we are ready to begin. In the main directory of the Project create a file. Name it whatever you have named your Project but in lowercase letters followed by *.rg* as extension.

E.g. if your Project was *Rentit* then your file would be *rentit.rg* This would be your main editor file for the RestGen DSL. When prompted by the IDE to select if you want to introduce xtext nature to the Project, select yes.

Start by declaring the *package*. This will be the base package of the application. Then declare the *dbconf*. After *dbconf* you can declare in any order one or more *internals* or *externals*. Xtext nature allows the Project to build on save. Once you save the *.rg* file, the entire Project will build and your files will be generated in the correct folders.

7 Conclusions

7.1 Work done so far

From the research it can be concluded that the questions from the problem statement have been answered.

- We have managed to find out what resource specific information is needed from the user when writing a Spring HATEOAS [6] application.
- We have managed to incorporate the structural and behavioural aspects [24] of a REST application into a model driven DSL input editor using Xtext. [14]
- Further managed to parse the information from it, that is needed for a Spring HATEOAS application. [6]
- Then managed to generate code from it by creating a generator with xtend. [38]
- We have managed to export the project as plugin for all eclipse based IDEs.
- We have tested a Spring boot application with HATEOAS with the Plant hire scenario [13] on Spring Tool Suite 3.7.2.RELEASE version.
- We have managed to make the plugin available as an Eclipse Project update site [15], by having the plugin available as a site on *github*. [41]

7.2 Future work

For the future there could be plenty of things to do. Some of the things in our minds are the following,

- Incorporate configuration for more databases
- Create a plugin for intellij idea.
- Incorporate functionality for generating resource querying.
- Diagrammatic representation with use of Ecore module of eclipse.
- Auto-generation of test classes for the same output code.
- Auto-generation of frontend views for the same resources.

8 References

[1] R. T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, Irvine, 2000.

[2] S. Schreier, „Modeling RESTful applications,“ 2011.

[3] L. Richardson, RESTful Web APIs, Sebastopol: O'Reilly Media, 2007.

[4] M. Messe, REST API Design Rulebook, Sebastopol: O'Reilly Media, 2011.

[5] „HTTP methods,“ HTTP, 2016. [Võrgumaterjal]. Available: http://www.w3schools.com/tags/ref_httpmethods.asp. [Kasutatud 9 May 2016].

[6] „Understanding HATEOAS,“ Pivotal software, 2016. [Võrgumaterjal]. Available: https://spring.io/understanding/HATEOAS. [Kasutatud 9 May 2016].

[7] „Spring framework,“ Pivotal software, 2016. [Võrgumaterjal]. Available: https://spring.io/. [Kasutatud 9 May 2016].

[8] „What is hypermedia,“ Smartbear software, 2016. [Võrgumaterjal]. Available: https://smartbear.com/learn/api-design/what-is-hypermedia/. [Kasutatud 9 May 2016].

[9] A. v. Deursen, Domain-Specific Languages: An Annotated Bibliography, Amsterdam: Sigplan Notices, 2000.

[10] „XML W3,“ W3 schools, 2016. [Võrgumaterjal]. Available: https://www.w3.org/XML/. [Kasutatud 9 May 2016].

[11] „Json home,“ Json, 2016. [Võrgumaterjal]. Available: http://www.json.org/. [Kasutatud 9 May 2016].

[12] „JVM,“ Oracle, 2016. [Võrgumaterjal]. Available: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-1.html#jvms-1.2. [Kasutatud 9 May 2016].

[13] „Enterprise System Intergration course page,“ 12 December 2015. [Võrgumaterjal]. Available: https://courses.cs.ut.ee/MTAT.03.229/2014_fall/uploads/Main/PlantHireScenario.pdf.

[14] „Xtext website,“ 13 Nov 2015. [Võrgumaterjal]. Available: https://eclipse.org/Xtext/index.html.

[15] „Eclipse Home,“ 2016. [Võrgumaterjal]. Available: https://www.eclipse.org/. [Kasutatud 18 May 2016].

[16] „UML,“ 2016. [Võrgumaterjal]. Available: http://www.uml.org/what-is-uml.htm. [Kasutatud 9 may 2016].

[17] „Eclipse Ecore,“ 2016. [Võrgumaterjal]. Available: http://www.eclipse.org/ecoretools/. [Kasutatud 18 May 2016].

[18] „Apiary Website,“ 13 Nov 2015. [Võrgumaterjal]. Available: https://apiary.io/how-it-works.

[19] „Swagger Website,“ 13 Nov 2015. [Võrgumaterjal]. Available: http://swagger.io/.

[20] „RAML website,“ 13 Nov 2015. [Võrgumaterjal]. Available: http://raml.org/.

[21] „RestUnited Website,“ 13 Nov 2015. [Võrgumaterjal]. Available: https://restunited.com/.

[22] „Restlet Studio,“ 13 Nov 2015. [Võrgumaterjal]. Available: http://studio.restlet.com/.

[23] P. Selonen, „Towards a Model-Driven Process for Designing ReSTful Web Services,“ 2009.

[24] Porres, „Modeling Behavioral RESTful Web Service Interfaces in UML,“ 2011.

[25] „Spring Boot application,“ 2015. [Võrgumaterjal]. Available: https://docs.spring.io/spring-boot/docs/current/reference/html/index.html. [Kasutatud 20 Nov 2015].

[26] „Java home,“ Oracle, 2016. [Võrgumaterjal]. Available: https://www.oracle.com/java/index.html. [Kasutatud 9 May 2016].

[27] „Postgres Sql Home,“ Postgres Sql, 2016. [Võrgumaterjal]. Available: http://www.postgresql.org/. [Kasutatud 9 May 2016].

[28] „Spring HTTPMethodType,“ Spring Framework, 2016. [Võrgumaterjal]. Available: https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/http/HttpMethod.html. [Kasutatud 9 May 2016].

[29] „intellij,“ 2016. [Võrgumaterjal]. Available: https://www.jetbrains.com/idea/. [Kasutatud 18 May 2016].

[30] „IDE,“ [Võrgumaterjal]. Available: http://searchsoftwarequality.techtarget.com/definition/integrated-development-environment. [Kasutatud 9 May 2016].

[31] „Eclipse downloads,“ 2016. [Võrgumaterjal]. Available: https://www.eclipse.org/downloads/. [Kasutatud 18 May 2016].

[32] „Xtext eclipse plugin,“ 2016. [Võrgumaterjal]. Available: https://marketplace.eclipse.org/content/xtext. [Kasutatud 18 May 2016].

[33] „Xtend plugin,“ 2016. [Võrgumaterjal]. Available: https://marketplace.eclipse.org/content/eclipse-xtend. [Kasutatud 18 May 2016].

[34] „Xtext idea plugin,“ 2016. [Võrgumaterjal]. Available: https://plugins.jetbrains.com/plugin/8072?pr=idea. [Kasutatud 18 May 2016].

[35] „xtext documentation,“ 2016. [Võrgumaterjal]. Available: https://eclipse.org/Xtext/documentation/301_grammarlanguage.html. [Kasutatud 18 May 2016].

[36] „Xtext Editor tutorial,“ 2016. [Võrgumaterjal]. Available: https://eclipse.org/Xtext/documentation/102_domainmodelwalkthrough.html. [Kasutatud 18 May 2016].

[37] „Output outlet for xtext,“ 2015. [Võrgumaterjal]. Available: http://stackoverflow.com/questions/10350022/how-to-add-multiple-outlets-for-generated-xtext-dsl. [Kasutatud 15 Nov 2015].

[38] „Xtext generator tutorial,“ [Võrgumaterjal]. Available: https://eclipse.org/Xtext/documentation/103_domainmodelnextsteps.html. [Kasutatud 18 May 2016].

[39] „xtext api docs,“ 2016. [Võrgumaterjal]. Available: http://download.eclipse.org/modeling/tmf/xtext/javadoc/2.9/. [Kasutatud 18 May 2016].

[40] „JVM and Import manager,“ 2016. [Võrgumaterjal]. Available: http://www.rcp-vision.com/1573/using-jvm-types-in-xtext-2-1-and-the-importmanager/?lang=en. [Kasutatud 13 Feb 2016].

[41] V. Desai, „RestGen update site,“ 2016. [Võrgumaterjal]. Available: http://vishalkirandesai.github.io/. [Kasutatud 18 May 2016].

[42] „spring initialzr,“ 2016. [Võrgumaterjal]. Available: https://start.spring.io/. [Kasutatud 18 May 2016].

[43] „Java for loop,“ 2016. [Võrgumaterjal]. Available: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html. [Kasutatud 18 May 2016].

Appendix

I. Code for RgDsl.xtext

```
grammar ee.ut.restgen.rgdsl.RgDsl with org.eclipse.xtext.xbase.Xtype
generate rgDsl "http://www.ut.ee/restgen/rgdsl/RgDsl"
ResourceModel :
        'package' name = QualifiedName
        'dbconf' '{' databaseConfiguration = DatabaseConfiguration '}'
        (elements += ResourceType)*
;
DatabaseConfiguration:
        'database' type = DBType
        'username' username = STRING
        'password' password = STRING
        'port' port = NUMBER
;
terminal NUMBER:
        ('0' .. '9')(NUMBER)*
;
DBType:
        mongo = 'MONGO' | postgres = 'POSTGRES' | mysql = 'MYSQL'
;
QualifiedName:
  ID ('.' ID)*
;
ResourceType:
        Internal | External
;
enum Cardinality:
        onetoone = "OneToOne" | onetomany = "OneToMany" | manytoone = "Many-ToOne"
;
External:
        'external' name = ID '{'
                (datatypes += DataType)*
        '}'
```

```
;
Internal:
	'internal' name = ID '{'
		(datatypes += DataType)*
		(internals += InternalDataType)*
		'states' '{'
			(defaultState = State)*
			(','states += State)*
		'}'
		'transitions' '{'
		(transtions += Transition)*
		'}'
	'}'
;
State:
	name = ID
;
Query:
	queryName = STRING 'taking' params = QueryParams 'on' rel = STRING 'giving' responseBody = DataType|Internal
;
QueryParams:
	firstParam = DataType
	(','otherParams += DataType)*
;
enum HTTPMethodType:
	get = "GET" | post = "POST" | put = "PUT" | delete = "DELETE" | patch = "PATCH"
;
Transition:
	fromState = [State] 'to' toState = [State] 'using' controllerMethod = STRING 'with' methodType = HTTPMethodType 'on' rel = STRING
;
DataType:
  dataType = JvmTypeReference name = ID
;
```

```
InternalDataType:
        cardinality = Cardinality internal = [Internal] name = ID
;
```

II. Code for RestGenOutputConfiguration.java

```
package ee.ut.restgen.rgdsl;

import java.util.Set;

import org.eclipse.xtext.generator.IFileSystemAccess;
import org.eclipse.xtext.generator.IOutputConfigurationProvider;
import org.eclipse.xtext.generator.OutputConfiguration;

import static com.google.common.collect.Sets.newHashSet;

public class RestGenOutputConfiguration implements IOutputConfigurationProvider {

        public final static String DEFAULT_OUTPUT_FINAL = "DEFAULT_OUT-
PUT_FINAL";

        /**
         * @return a set of {@link OutputConfiguration} available for the generator
         */
        public Set<OutputConfiguration> getOutputConfigurations() {
          OutputConfiguration defaultOutput = new OutputConfiguration(IFileSystem-
Access.DEFAULT_OUTPUT);
          defaultOutput.setDescription("Output Folder");
          defaultOutput.setOutputDirectory("./src-gen");
          defaultOutput.setOverrideExistingResources(true);
          defaultOutput.setCreateOutputDirectory(true);
          defaultOutput.setCleanUpDerivedResources(true);
          defaultOutput.setSetDerivedProperty(true);

          OutputConfiguration onceOutput = new OutputConfiguration(DEFAULT_OUT-
PUT_FINAL);
          onceOutput.setDescription("Output Folder");
```

```
            onceOutput.setOutputDirectory("./src/main/java");
            onceOutput.setOverrideExistingResources(true);
            onceOutput.setCreateOutputDirectory(true);
            onceOutput.setCleanUpDerivedResources(false);
            onceOutput.setSetDerivedProperty(true);
            return newHashSet(defaultOutput, onceOutput);
        }

}
```

III. Code for RgDslGenerator.xtend

```
/*
 * generated by Xtext
 */
package ee.ut.restgen.rgdsl.generator

import com.google.inject.Inject
import ee.ut.restgen.rgdsl.rgDsl.External
import ee.ut.restgen.rgdsl.rgDsl.Internal
import ee.ut.restgen.rgdsl.rgDsl.ResourceModel
import javax.xml.bind.annotation.XmlRootElement
import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.xtext.generator.IFileSystemAccess
import org.eclipse.xtext.generator.IGenerator
import org.eclipse.xtext.naming.IQualifiedNameProvider
import org.eclipse.xtext.naming.QualifiedName
import org.eclipse.xtext.xbase.compiler.ImportManager
import ee.ut.restgen.rgdsl.rgDsl.DBType
import ee.ut.restgen.rgdsl.rgDsl.Cardinality

/**
 * Generates code from your model files on save.
 *
 * see http://www.eclipse.org/Xtext/documentation.html#TutorialCodeGeneration
 */
class RgDslGenerator implements IGenerator {
```

```
@Inject extension IQualifiedNameProvider

 override void doGenerate(Resource resource, IFileSystemAccess fsa) {
        var basePackage = "";
  for(rm: resource.allContents.toIterable.filter(ResourceModel)) {
        basePackage = rm.fullyQualifiedName.toString("/");
         fsa.generateFile(
     basePackage + "/" + rm.fullyQualifiedName.lastSegment.toFirstUpper+ "Application.java", "DEFAULT_OUTPUT_FINAL",
     rm.fullyQualifiedName.createApplication)
    fsa.generateFile(
     basePackage + "/" + rm.fullyQualifiedName.lastSegment.toFirstUpper+ "DatabaseConfiguration.java", "DEFAULT_OUTPUT_FINAL",
     rm.createConfiguration)
  }

  for(e: resource.allContents.toIterable.filter(Internal)) {
    fsa.generateFile(
     basePackage + "/rest/utils/ExtendedLink.java", "DEFAULT_OUTPUT_FINAL",
     e.createExtendedLinkSupport)
    fsa.generateFile(
     basePackage + "/rest/utils/ResourceSupport.java", "DEFAULT_OUTPUT_FINAL",
     e.createResourceSupport)
    fsa.generateFile(
     basePackage + "/models/" + e.name +".java", "DEFAULT_OUTPUT_FINAL",
     e.createEntity)
    fsa.generateFile(
     basePackage + "/repositories/" + e.name + "Repository.java", "DEFAULT_OUTPUT_FINAL",
     e.createRepository)
    fsa.generateFile(
     basePackage + "/rest/exceptions/" + e.name + "NotFoundException.java", "DEFAULT_OUTPUT_FINAL",
     e.createResourceException)
         fsa.generateFile(
```

```
      basePackage + "/rest/utils/" + e.name + "ResourceAssembler.java", "DE-
FAULT_OUTPUT_FINAL",
       e.createResourceAssembler)
     fsa.generateFile(
       basePackage + "/rest/" + e.name + "Resource.java", "DEFAULT_OUTPUT_FINAL",
       e.createResource)
     fsa.generateFile(
       basePackage + "/rest/controllers/" + e.name + "RestController.java", "DE-
FAULT_OUTPUT_FINAL",
       e.createRestController)
     fsa.generateFile(
       basePackage + "/models/" + e.name + "Status.java", "DEFAULT_OUTPUT_FINAL",
       e.createStatus)
   }

   for(e: resource.allContents.toIterable.filter(External)) {
     fsa.generateFile(
       basePackage + "/rest/" + e.name + "Resource.java",  "DEFAULT_OUTPUT_FINAL",
       e.createResource)
   }
 }

// Application part
 def createApplication(QualifiedName basePackage) '"
package «basePackage.toString»;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class «basePackage.lastSegment.toFirstUpper»Application {

   public static void main(String[] args) {
      SpringApplication.run(«basePackage.lastSegment.toFirstUpper»Application.class,
args);
   }
```

```
}

 '''

// Configuration part
def createConfiguration(ResourceModel rm) '''
package «rm.fullyQualifiedName.toString»;

import java.net.URI;
import java.net.URISyntaxException;

import javax.sql.DataSource;

import org.apache.commons.dbcp.BasicDataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class «rm.fullyQualifiedName.lastSegment.toFirstUpper»DatabaseConfiguration {

   @Bean
   public DataSource dataSource()  {

      URI dbUri;
      try {
         String username = "«rm.databaseConfiguration.username»";
         String password = "«rm.databaseConfiguration.password»";
         String url = "jdbc:postgresql://localhost:«rm.databaseConfiguration.port»/«rm.fullyQualifiedName.lastSegment»-test";

         String dbProperty = System.getenv("DATABASE_URL");
         if(dbProperty != null) {
            dbUri = new URI(dbProperty);

            username = dbUri.getUserInfo().split(":")[0];
```

```
            password = dbUri.getUserInfo().split(":")[1];
            url = "jdbc:postgresql://" + dbUri.getHost() + ':' + dbUri.getPort() + dbUri.getPath();
        }

        BasicDataSource basicDataSource = new BasicDataSource();
        basicDataSource.setUrl(url);
        basicDataSource.setUsername(username);
        basicDataSource.setPassword(password);

        return basicDataSource;

      } catch (URISyntaxException e) {
        return null;
      }
   }
}
'''

// Extended link part
 def createExtendedLinkSupport(Internal e) '''
package «e.fullyQualifiedName.skipLast(1)».rest.utils;

import javax.xml.bind.annotation.XmlType;

import org.springframework.hateoas.Link;

@XmlType(name = "_link", namespace = Link.ATOM_NAMESPACE)
public class ExtendedLink extends Link {
        private static final long serialVersionUID = -9037755944661782122L;
        private String method;

        protected ExtendedLink(){}

        public ExtendedLink(String href, String rel, String method){
```

```
		super(href, rel);
		this.method = method;
	}

	public String getMethod(){
		return method;
	}

	public void setMethod(String method){
		this.method = method;
	}
}

 '''

// Resource support part
 def createResourceSupport(Internal e) '''
package «e.fullyQualifiedName.skipLast(1)».rest.utils;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlTransient;

import org.codehaus.jackson.annotate.JsonProperty;
import org.springframework.hateoas.Link;

@XmlTransient
public class ResourceSupport extends org.springframework.hateoas.ResourceSupport{
	@XmlElement(name = "_link", namespace = Link.ATOM_NAMESPACE)
	@JsonProperty("_links")
	private final List<ExtendedLink> _links;
```

```
	public ResourceSupport(){
		super();
		this._links = new ArrayList<>();
	}

	public void add(Link link) {
		if(link instanceof ExtendedLink)
			this._links.add((ExtendedLink) link);
		else
			super.add(link);
	}

	public List<ExtendedLink> get_links() {
		return Collections.unmodifiableList(_links);
	}

	public void remove_links() {
		_links.clear();
	}

	public Link get_link(String rel) {

		for (Link link : _links) {
			if (link.getRel().equals(rel)) {
				return link;
			}
		}

		return null;
	}
}

 '''

// Entity part
```

```
 def createEntity(Internal e) '''
package «e.fullyQualifiedName.skipLast(1)».models;

  «val importManager = new ImportManager(true)»
  «val mainMethod = compileEntity(e, importManager)»
«IF !importManager.imports.empty»
        «FOR i:importManager.imports»
import «i.toString»;
        «ENDFOR»
«ENDIF»
«FOR i:e.internals»
import «i.internal.fullyQualifiedName.skipLast(1)».models.«i.internal.name»;
import javax.persistence.«i.cardinality»;
«ENDFOR»
import javax.persistence.Entity;
import javax.persistence.Enumerated;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import static javax.persistence.EnumType.STRING;
«mainMethod»
  '''

def compileEntity(Internal e, ImportManager manager) '''

@Entity
public class «e.name» {

        @Id
   @GeneratedValue
   Long id;

   public Long getId() {
      return id;
   }
```

```
    @Enumerated(STRING)
    «e.name»Status «e.name.toFirstLower»Status;

    public «e.name»Status get«e.name»Status() {
       return «e.name.toFirstLower»Status;
    }

    public void set«e.name»Status(«e.name»Status «e.name.toFirstLower»Status) {
       this.«e.name.toFirstLower»Status = «e.name.toFirstLower»Status;
    }

    «FOR i:e.internals»
          @«i.cardinality»
          «i.internal.name» «i.internal.name.toFirstLower»;

          public void set«i.internal.name.toFirstUpper»(«i.internal.name» «i.name») {
                    this.«i.name» =  «i.name»;
                }

                public «i.internal.name» get«i.name.toFirstUpper»() {
                    return «i.name»;
                }
    «ENDFOR»

          «FOR f:e.datatypes»
                «manager.serialize(f.dataType.type)» «f.name»;

                public void set«f.name.toFirstUpper»(«f.dataType.simpleName» «f.name»)
{
                    this.«f.name» =  «f.name»;
                }

                public «f.dataType.simpleName» get«f.name.toFirstUpper»() {
                    return «f.name»;
                }
```

```
        «ENDFOR»
}
'''

// Controller part
  def createRestController(Internal e) '''
package «e.fullyQualifiedName.skipLast(1)».rest.controllers;

  «val importManager = new ImportManager(true)»
  «val mainMethod = compileRestController(e, importManager)»
  «IF !importManager.imports.empty»
        «FOR i:importManager.imports»
import «i»;
        «ENDFOR»
«ENDIF»
import «e.fullyQualifiedName.skipLast(1)».models.«e.name»;
import «e.fullyQualifiedName.skipLast(1)».repositories.«e.name»Repository;
import «e.fullyQualifiedName.skipLast(1)».rest.«e.name»Resource;
import «e.fullyQualifiedName.skipLast(1)».rest.utils.«e.name»ResourceAssembler;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

import static «e.fullyQualifiedName.skipLast(1)».models.«e.name»Status.*;
import static org.springframework.web.bind.annotation.RequestMethod.DELETE;
import static org.springframework.web.bind.annotation.RequestMethod.GET;
import static org.springframework.web.bind.annotation.RequestMethod.POST;
import static org.springframework.web.bind.annotation.RequestMethod.PUT;
import static org.springframework.web.bind.annotation.RequestMethod.PATCH;

«mainMethod»
```

```
 '''

 def compileRestController(Internal e, ImportManager manager) '''
@RestController
@RequestMapping(value = "«e.name.toLowerCase»")
public class «e.name»RestController {

   @Autowired
   «e.name»Repository «e.name.toFirstLower»Repository;
«FOR internal: e.internals»
         «internal.name»ResourceAssembler «internal.name.toFirstLower»ResourceAssembler =
         new «internal.name»ResourceAssembler();
«ENDFOR»
   «e.name»ResourceAssembler «e.name.toFirstLower»ResourceAssembler =
         new «e.name»ResourceAssembler();

   @RequestMapping(method = GET)
   public List<«e.name»Resource> get«e.name»s() {
      List<«e.name»> «e.name.toFirstLower»s = «e.name.toFirstLower»Repository.findAll();
      return «e.name.toFirstLower»ResourceAssembler.toResources(«e.name.toFirstLower»s);
   }

   @RequestMapping(method = GET, value = "{id}")
   public «e.name»Resource get«e.name»(@PathVariable Long id) {
      «e.name» «e.name.toFirstLower» = «e.name.toFirstLower»Repository.findOne(id);
      return «e.name.toFirstLower»ResourceAssembler.toResource(«e.name.toFirstLower»);
   }

«FOR transition: e.transtions»
         @RequestMapping(method = «transition.methodType», value = "{id}/«transition.rel»")
   public «e.name»Resource «transition.controllerMethod.toFirstLower»(Long id) {
      «e.name» «e.name.toFirstLower» = «e.name.toFirstLower»Repository.findOne(id);
```

```
    «e.name.toFirstLower».set«e.name»Status(«transition.toState.name»);
    «e.name.toFirstLower»Repository.saveAndFlush(«e.name.toFirstLower»);
    return «e.name.toFirstLower»ResourceAssembler.toResource(«e.name.toFirst-
Lower»);
  }
«ENDFOR»

«FOR internal: e.internals»
«IF internal.cardinality.equals(Cardinality.ONETOONE)»
      @RequestMapping(method = PUT, value = "{id}/«internal.name.toLowerCase»")
      public ResponseEntity<Void> modify«internal.name.toFirstUpper»(@PathVaria-
ble Long id, @RequestBody «internal.name.toFirstUpper»Resource «internal.name.toFirst-
Lower»Resource) {
    «e.name» «e.name.toFirstLower» = «e.name.toFirstLower»Repository.findOne(id);
    «internal.name» «internal.name.toFirstLower» = «e.name.toFirstLower».get«inter-
nal.name.toFirstUpper»();
          //TODO: write the modification lines.
          «e.name.toFirstLower»Repository.saveAndFlush(«e.name.toFirstLower»);
          return new ResponseEntity<Void>(HttpStatus.OK);

      @RequestMapping(method = GET, value = "{id}/«internal.name.toLowerCase»")
      public ResponseEntity<Void> modify«internal.name.toFirstUpper»(@PathVaria-
ble Long id) {
    «e.name» «e.name.toFirstLower» = «e.name.toFirstLower»Repository.findOne(id);
    «internal.name» «internal.name.toFirstLower» = «e.name.toFirstLower».get«inter-
nal.name.toFirstUpper»();
          return «internal.name.toFirstLower»ResourceAssem-
bler.toResource(«e.name.toFirstLower»);
«ELSEIF internal.cardinality.equals(Cardinality.ONETOMANY)»

«ENDIF»
«ENDFOR»
}
 '''

// Resource part
 def createResource(Internal e) '''
```

```
package «e.fullyQualifiedName.skipLast(1)».rest;

  «val importManager = new ImportManager(true)»
  «val mainMethod = compileResource(e, importManager)»
«IF !importManager.imports.empty»
        «FOR i:importManager.imports»
import «i»;
        «ENDFOR»
«ENDIF»
«FOR i:e.internals»
import «i.internal.fullyQualifiedName.skipLast(1)».rest.«i.internal.name»Resource;
«ENDFOR»
import «e.fullyQualifiedName.skipLast(1)».rest.utils.ResourceSupport;

«mainMethod»
  '''

  def createResource(External e) '''
package «e.fullyQualifiedName.skipLast(1)».rest;

  «val importManager = new ImportManager(true)»
  «val mainMethod = compileResource(e, importManager)»
«IF !importManager.imports.empty»
        «FOR i:importManager.imports»
import «i»;
        «ENDFOR»
«ENDIF»
import «e.fullyQualifiedName.skipLast(1)».rest.utils.ResourceSupport;

«mainMethod»
  '''

def compileResource(Internal e, ImportManager manager) '''
@«manager.serialize(XmlRootElement)»(name="«e.name.toLowerCase»")
public class «e.name»Resource extends ResourceSupport{
```

```
	«FOR i:e.internals»
	«i.internal.name»Resource «i.internal.name.toFirstLower»Resource;

	public void set«i.internal.name.toFirstUpper»Resource(«i.internal.name»Resource «i.name»Resource) {
		this.«i.name»Resource =  «i.name»Resource;
	}

	public «i.internal.name»Resource get«i.name.toFirstUpper»Resource() {
		return «i.name»Resource;
	}
  «ENDFOR»

	«FOR f:e.datatypes»
		«manager.serialize(f.dataType.type)» «f.name»;

		public void set«f.name.toFirstUpper»(«f.dataType.simpleName» «f.name») {
			this.«f.name» =  «f.name»;
		}

		public «f.dataType.simpleName» get«f.name.toFirstUpper»() {
			return «f.name»;
		}

	«ENDFOR»
}
'''

def compileResource(External e, ImportManager manager) '''
@«manager.serialize(XmlRootElement)»(name="«e.name.toLowerCase»")
public class «e.name»Resource extends ResourceSupport{
	«FOR f:e.datatypes»
		«manager.serialize(f.dataType.type)» «f.name»;
```

```
		public void set«f.name.toFirstUpper»(«f.dataType.simpleName» «f.name»)
{
			this.«f.name» =  «f.name»;
		}

		public «f.dataType.simpleName» get«f.name.toFirstUpper»() {
			return «f.name»;
		}

	«ENDFOR»
}
'''

// Status part
def createStatus(Internal e) '''
package «e.fullyQualifiedName.skipLast(1)».models;

public enum «e.name»Status {
	«IF e.defaultState != null»
	«e.defaultState»,
   «FOR state: e.states»
   «state.name»,
   «ENDFOR»
   «ENDIF»
}
'''

// Repository part
  def createRepository(Internal e) '''
package «e.fullyQualifiedName.skipLast(1)».repositories;

  «val importManager = new ImportManager(true)»
  «val mainMethod = compileRepository(e, importManager)»
«IF !importManager.imports.empty»
	«FOR i:importManager.imports»
```

```
import «i»;
        «ENDFOR»
«ENDIF»
import «e.fullyQualifiedName.skipLast(1)».models.«e.name»;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
«mainMethod»
  '''

def compileRepository(Internal e, ImportManager manager) '''

@Repository
public interface «e.name»Repository extends JpaRepository<«e.name», Long> {
}
'''

// ResourceAssembler part
  def createResourceAssembler(Internal e) '''
package «e.fullyQualifiedName.skipLast(1)».rest.utils;

  «val importManager = new ImportManager(true)»
  «val mainMethod = compileResourceAssembler(e, importManager)»
«IF !importManager.imports.empty»
        «FOR i:importManager.imports»
import «i»;
        «ENDFOR»
«ENDIF»
import «e.fullyQualifiedName.skipLast(1)».models.«e.name»;
import «e.fullyQualifiedName.skipLast(1)».rest.«e.name»Resource;
import «e.fullyQualifiedName.skipLast(1)».rest.controllers.«e.name»RestController;
import org.springframework.hateoas.mvc.ResourceAssemblerSupport;
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.linkTo;
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.methodOn;

«mainMethod»
```

```
'''

def compileResourceAssembler(Internal e, ImportManager manager) '''
public class «e.name»ResourceAssembler extends ResourceAssemblerSupport<«e.name», «e.name»Resource>{

   public «e.name»ResourceAssembler() {
      super(«e.name»RestController.class, «e.name»Resource.class);
   }

   @«manager.serialize(Override)»
   public «e.name»Resource toResource(«e.name» «e.name.toFirstLower») {
      «e.name»Resource «e.name.toFirstLower»Resource = createResourceWithId(«e.name.toFirstLower».getId(), «e.name.toFirstLower»);
      «FOR type:e.datatypes»
      «e.name.toFirstLower»Resource.set«type.name.toFirstUpper»(«e.name.toFirstLower».get«type.name.toFirstUpper»());
      «ENDFOR»

      switch («e.name.toFirstLower».get«e.name»Status()) {
         «FOR state: e.states»
         case «state.name»:
         «FOR transition: e.transtions»
         «IF state == transition.fromState»
                  «e.name.toFirstLower»Resource.add(
                  new ExtendedLink(linkTo(methodOn(«e.name»RestController.class)
                        .«transition.controllerMethod.toFirstLower»(«e.name.toFirstLower».getId()))
                        .toString(), "«transition.rel»", "«transition.methodType»"));
         «ENDIF»
         «ENDFOR»
                  break;
         «ENDFOR»
         default:
            break;
      }
```

```
        return «e.name.toFirstLower»Resource;
    }
}
'''

// ResourceException part
  def createResourceException(Internal e) '''
package «e.fullyQualifiedName.skipLast(1)».rest.exceptions;

public class «e.name»NotFoundException extends Exception {
    private static final long serialVersionUID = 1L;

    public «e.name»NotFoundException(Long id) {
        super(String.format("«e.name» not found! («e.name» id: %d)", id));
    }
}
'''

}
```

IV. Code for PurchaseOrderResource.java

```
package buildit.rest;
import buildit.rest.utils.ResourceSupport;
import javax.xml.bind.annotation.XmlRootElement;
import java.util.Date;

@XmlRootElement(name = "purchaseorder")
public class PurchaseOrderResource extends ResourceSupport{
    String contact;
    Long startDate;
    Long endDate;
    float price;

    public String getContact() {
```

```
        return contact;
    }
    public void setContact(String contact) {
        this.contact = contact;
    }
    public Long getStartDate() {
        return startDate;
    }
    public void setStartDate(Long startDate) {
        this.startDate = startDate;
    }
    public Long getEndDate() {
        return endDate;
    }
    public void setEndDate(Long endDate) {
        this.endDate = endDate;
    }
    public float getPrice() {
        return price;
    }

    public void setPrice(float price) {
        this.price = price;
    }
}
```

V. Code for PurchaseOrderResourceAssembler.java

```
package buildit.rest.utils;

import buildit.models.PurchaseOrder;
import buildit.rest.PurchaseOrderResource;
import buildit.rest.controllers.PurchaseOrderRestController;
import org.springframework.hateoas.mvc.ResourceAssemblerSupport;
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.linkTo;
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.methodOn;

public class PurchaseOrderResourceAssembler extends ResourceAssemblerSupport<PurchaseOrder, PurchaseOrderResource>{

   public PurchaseOrderResourceAssembler() {
      super(PurchaseOrderRestController.class, PurchaseOrderResource.class);
   }

   @Override
   public PurchaseOrderResource toResource(PurchaseOrder purchaseOrder) {
      PurchaseOrderResource purchaseOrderResource = createResourceWithId(purchaseOrder.getId(), purchaseOrder);
      purchaseOrderResource.setContact(purchaseOrder.getCustomer().getName());
      purchaseOrderResource.setStartDate(purchaseOrder.getStartDate().getTimeInMillis());
      purchaseOrderResource.setEndDate(purchaseOrder.getEndDate().getTimeInMillis());
      purchaseOrderResource.setPrice(purchaseOrder.getPrice());

      switch (purchaseOrder.getPoStatus()) {
         case PENDING_CONFIRMATION:
            purchaseOrderResource.add(
                  new ExtendedLink(linkTo(methodOn(PurchaseOrderRestController.class)
                        .rejectPO(purchaseOrder.getId()))
                        .toString(), "rejectPO", "DELETE"));
            purchaseOrderResource.add(
                  new ExtendedLink(linkTo(methodOn(PurchaseOrderRestController.class)
```

```
                        .acceptPO(purchaseOrder.getId()))
                        .toString(), "acceptPO", "POST"));
                break;
            case OPEN:
                purchaseOrderResource.add(
                    new ExtendedLink(linkTo(methodOn(PurchaseOrderRestController.class)
                        .closePO(purchaseOrder.getId()))
                        .toString(), "closePO", "DELETE"));
            default:
                break;
        }

        return purchaseOrderResource;
    }
}
```

VI. Code for PurchaseOrderRestController.java

```
package buildit.rest.controllers;

import buildit.models.PurchaseOrder;
import buildit.repositories.PurchaseOrderRepository;
import buildit.rest.PurchaseOrderResource;
import buildit.rest.utils.PurchaseOrderResourceAssembler;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import java.util.List;
import static buildit.models.POStatus.CLOSED;
import static buildit.models.POStatus.OPEN;
import static buildit.models.POStatus.REJECTED;
import static org.springframework.web.bind.annotation.RequestMethod.DELETE;
import static org.springframework.web.bind.annotation.RequestMethod.GET;
import static org.springframework.web.bind.annotation.RequestMethod.POST;

@RestController
@RequestMapping(value = "po")
public class PurchaseOrderRestController {

   @Autowired
   PurchaseOrderRepository purchaseOrderRepository;

   PurchaseOrderResourceAssembler purchaseOrderResourceAssembler =
         new PurchaseOrderResourceAssembler();

   @RequestMapping(method = GET)
   public List<PurchaseOrderResource> getPos() {
      List<PurchaseOrder> pos = purchaseOrderRepository.findAll();
      return purchaseOrderResourceAssembler.toResources(pos);
   }
```

```
    @RequestMapping(method = GET, value = "{id}")
    public PurchaseOrderResource getPo(@PathVariable Long id) {
        PurchaseOrder purchaseOrder = purchaseOrderRepository.findOne(id);
        return purchaseOrderResourceAssembler.toResource(purchaseOrder);
    }


    @RequestMapping(method = DELETE, value = "{id}/acceptance")
    public PurchaseOrderResource rejectPO(Long id) {
        PurchaseOrder purchaseOrder = purchaseOrderRepository.findOne(id);
        purchaseOrder.setPoStatus(REJECTED);
        purchaseOrderRepository.saveAndFlush(purchaseOrder);
        return purchaseOrderResourceAssembler.toResource(purchaseOrder);
    }

    @RequestMapping(method = POST, value = "{id}/acceptance")
    public PurchaseOrderResource acceptPO(Long id) {
        PurchaseOrder purchaseOrder = purchaseOrderRepository.findOne(id);
        purchaseOrder.setPoStatus(OPEN);
        purchaseOrderRepository.saveAndFlush(purchaseOrder);
        return purchaseOrderResourceAssembler.toResource(purchaseOrder);
    }

    @RequestMapping(method = DELETE, value = "{id}/closure")
    public PurchaseOrderResource closePO(Long id) {
        PurchaseOrder purchaseOrder = purchaseOrderRepository.findOne(id);
        purchaseOrder.setPoStatus(CLOSED);
        purchaseOrderRepository.saveAndFlush(purchaseOrder);
        return purchaseOrderResourceAssembler.toResource(purchaseOrder);
    }
}
```

VII. Code for POStatus.java

```
package buildit.models;

public enum POStatus {
   PENDING_CONFIRMATION,
   OPEN,
   REJECTED,
   CLOSED,
   PENDING_UPDATE
}
```

VIII. Code for example_module.rg

```
package ee.ut.rentit

dbconf {
	database POSTGRES
	username 'postgres'
	password 'letmein'
	port 5432
}
internal PurchaseOrder {

	java.util.Date startDate
	java.util.Date endDate

	OneToOne Plant plant

	states {
		OPEN,
		APPROVED,
		DENIED,
		ACCEPTED,
		REJECTED,
		CLOSED
	}

	transitions {
		OPEN to APPROVED using 'approvePO' with POST on 'approval'
		OPEN to DENIED using 'denyPO' with DELETE on 'approval'
		DENIED to OPEN using 'reopenPO' with PUT on 'reopen'
		APPROVED to ACCEPTED using 'acceptPO' with POST on 'acceptance'
		APPROVED to REJECTED using 'rejectPO' with DELETE on 'acceptance'
		REJECTED to APPROVED using 'reApprovePO' with PUT on 'reapprove'
		ACCEPTED to CLOSED using 'closePO' with DELETE on 'closure'
	}
}

internal Invoice {
	java.lang.Float total

	OneToOne PurchaseOrder purchaseOrder

	states {
		OPEN,
		UNPAID,
		PAID
	}
```

```
        transition OPEN to UNPAID using 'sendInvoice' with POST on 'send'
        transition UNPAID to PAID using 'acceptPayment' with POST on 'payment'
}
```

IX. Code for RestGenOutputConfiguration.java

```
package ee.ut.restgen.rgdsl;

import java.util.Set;

import org.eclipse.xtext.generator.IFileSystemAccess;
import org.eclipse.xtext.generator.IOutputConfigurationProvider;
import org.eclipse.xtext.generator.OutputConfiguration;

import static com.google.common.collect.Sets.newHashSet;

public class RestGenOutputConfiguration implements IOutputConfigurationProvider {

	public final static String DEFAULT_OUTPUT_FINAL = "DEFAULT_OUT-
PUT_FINAL";

	/**
	 * @return a set of {@link OutputConfiguration} available for the generator
	 */
	public Set<OutputConfiguration> getOutputConfigurations() {
	  OutputConfiguration defaultOutput = new OutputConfiguration(IFileSystem-
Access.DEFAULT_OUTPUT);
	  defaultOutput.setDescription("Output Folder");
	  defaultOutput.setOutputDirectory("./src-gen");
	  defaultOutput.setOverrideExistingResources(true);
	  defaultOutput.setCreateOutputDirectory(true);
	  defaultOutput.setCleanUpDerivedResources(true);
	  defaultOutput.setSetDerivedProperty(true);

	  OutputConfiguration onceOutput = new OutputConfiguration(DEFAULT_OUT-
PUT_FINAL);
	  onceOutput.setDescription("Output Folder");
	  onceOutput.setOutputDirectory("./src/main/java");
	  onceOutput.setOverrideExistingResources(false);
	  onceOutput.setCreateOutputDirectory(true);
```

```
            onceOutput.setCleanUpDerivedResources(false);
            onceOutput.setSetDerivedProperty(true);
            return newHashSet(defaultOutput, onceOutput);
        }

}
```

X. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Vishal** (date of birth: 01.10.1988), herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Model driven engineering of Hypermedia REST applications,

supervised by Luciano García-Bañuelos,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **19.05.2016**