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:

Many tools have been developed that generate the skeleton of a basic REST-based application following the model-view-controller design pattern. However, little attention has been paid to developing tools that support Hypermedia-enabled applications, despite the increased interest shown by the software industry to this style of applications. 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 required is a target framework for generation of the code. To begin with, 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. Firstly, there is a need to know what kind of inputs or models would be required. REST modelling consists of two parts: structural modelling and behavioural modelling. Structural modelling is usually done with class diagrams while behavioural modelling is usually done with state charts. The output generation part has to be developed in a manner that it would allow, in future, to generate code for various languages. This would serve as a guideline for future work. In this paper, we introduce RestGen, a simple, intuitive yet powerful domain specific language (DSL) that helps developers to specify a REST API and that generates the skeleton of a Spring-based Java application that complies with the intended API. The DSL has been implemented as an Eclipse plugin, which demonstrates the feasibility of the approach.

Keywords: REST, Spring HATEOAS, Hypermedia, Code generation, Xtext

CERCS: P175

Mudelipõhine Hypermedia REST rakenduste programmeerimine

Lühikokkuvõte:

On välja töötatud mitmeid töövahendeid, mis genereerivad elementaarse REST-põhise rakenduse kondikava järgides model-view-controller disainimustrit. Ent on pööratud vähe tähelepanu arendamaks töövahendeid, mis toetavad Hypermedia poolt lubatud rakendusi, vaatamata tarkvaratööstuse poolt näidatud huvi kasvule seda tüüpi rakenduste vastu. Selle uurimuse eesmärk on leida kindel, teostatav ja efektiivne lahendus, et võtta sisendina rakenduse struktuursed ja käitumuslikud REST mudelid ja genereerida hüpermeedia REST rakenduse programmeerimise liidese kondikava. Vajalik on sihtraamistik koodi genereerimiseks. Alustuseks on vajalik keskenduda Javale Spring Boot raamistikuga ja Spring MVC struktuuriga. Selle uurimuse ulatus piirdub ainult Java keelega. Hiljem on võimalik laieneda ka teistele keeltele. Esmalt on tarvis teada, missuguseid sisendeid ja mudeleid oleks vaja. REST modelleerimine koosneb kahest osast – struktuurne modelleerimine ja käitumuslik modelleerimine. Struktuurset modelleerimist viiakse tavaliselt läbi klassidiagrammidega, samas kui käitumuslikku modelleerimist teostatakse seisunditabelitega. Väljundi genereerimise osa peab olema välja töötatud viisil, mis lubaks tulevikus genereerida koodi erinevate keelte jaoks. See oleks suuniseks tööde jaoks tulevikus. Selles töös me tutvustame RestGeni – lihtsat, intuitiivset, kuid võimast domeenispetsiifilist keelt (ingl domain specific language, DSL), mis aitab arendajatel määratleda RESTi rakendusliidest API (ingl Application Programming Interface) ja genereerib Spring-põhise Java rakenduse, mis ühildub ettenähtud APIga. DSL on kasutusele võetud Eclipse pluginina, mis demonstreerib antud meetodi teostatavust.

Võtmesõnad: REST, Spring HATEOAS, Hypermedia, Koodi genereerimine, Xtext

CERCS: P175

Table of Contents

1 Introduction

1.1 Motivation

REST applications have become popular over the years which has led to many tools having been created to generate REST applications. However, enough attention has not been paid to address Hypermedia based REST applications. These applications are increasingly becoming popular and the interest for developing code generation tools for these applications has increased over the years.

The objective of this research is to come up with a solid, feasible and efficient solution to take, as inputs, the structural and the behavioural aspects and then generate the basic skeleton code for the REST [1] application along with its Hypermedia support.

1.2 Problem Statement

REST [1] modelling consists of two parts: the structural modelling and behavioural modelling [2]. Structural modelling is a product of class diagrams of an application whereas, behavioural modelling is a product of the state chart of the dynamic classes of the application. With the use of these models, an application can be represented showing its REST [1] properties.

It is possible to design a Hypermedia-enabled REST [1] application using the structural and the behavioural REST models. Thereafter, these models are interpreted by a developer with the aim of gathering information necessary to write the code. With a predefined scope and a target framework, we can select the specific information necessary to generate the code. Then, a tool can be created to allow the user to specify the structural and behavioural aspects of the application in such a way that it provides the selected information necessary to generate the code. Therefore, this paper aims at answering the following questions:

- How can we generate the skeleton of a Hypermedia REST application by taking the required inputs from the user?
  - What kind of inputs, in general, are required to generate a REST application with Hypermedia support?
  - 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 provided to generate the end code?

1.3 Procedure

Studying the current state of the art, it should take into account the limitations as well as the potential plus points of existing technology. Knowing the possibility of existing project creation procedures, this paper would aim at achieving results that would enhance the overall project creation process by including Hypermedia [3] support. The following would qualify as the milestones for this research:

- Identify information required from the user when creating a Hypermedia REST application.
- Create a syntax to get the above-mentioned information from the user while modelling and then create a suitable DSL editor using the syntax.
- Create a code generation tool that would take the information provided by the user in the editor and generate the Hypermedia REST application.

2 Background

2.1 Common Words and Concepts

Before moving further into the paper, one must understand some vocabulary associated with the domain. Some of these are concepts are essential to the understanding of the research and the further sections of this paper.

2.1.1 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.

2.1.2 Application Programming Interface (API)

A detailed documentation of a software component enlisting its methods, properties, hierarchy, inheritance, and visibility.

2.1.3 Domain-specific language (DSL)

A problem-oriented programming language that consists of abstractions and notations specific towards a solution to the problem rather than a generic form. [4]

2.1.4 XML [5]

An extensible markup language is a flexible hierarchical text format which is used widely as a standard for data transmission payload format.

2.1.5 JSON [6]

JavaScript Object Notion is a data transmission and interchange format widely used owing to its simple parsable format which allows its use with a variety of programming languages.

2.1.6 JVM [7]

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

2.1.7 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.

2.1.7.1 Web services and Resource Oriented Architecture

Restful web services by Richardson et al [8] states the difference between REST and resource oriented architecture. Though it might sound the same but REST is a set of guidelines and resource oriented architecture is more specific.

Resource oriented architecture [8] is the architectural design of a service, based on individual perception and individual understanding of REST concepts. Every web service providing REST services, having its own resource oriented architecture, has some common REST concepts.

Every organisation would have its own architecture. This would definitely be RESTful if the guidelines are followed but they may not necessarily be the same. There is a great variety of choice with which one could design the architecture of the web service using the REST guidelines.

2.1.7.2 Resource

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

When many resources can be categorised as a particular type, then we come across the concept of a container. A container is a collection of all resources of the same type. Every container will contain resources that have similar properties, but different identities. For example, *dogs* will qualify as the container and a *dog* named 'Jimmy' will qualify as a resource of that container.

2.1.7.3 REST API design

Masse et al REST API design rulebook [9] provides some very useful guidelines for designing the API and the URIs.

- **Resource Identifier and URI**

  Every resource is uniquely identified by a Resource Identifier [1]. The service provided this resource can be reached by the Uniform Resource Identifier (URI) [1] of the resource. For a single unique resource, its URI and its Resource Identifier are both unique. This enables service provision directly related to the resource involved.

  Every resource, as we know, is identified by a Resource Identifier and addressed using a Uniform Resource Identifier [1]. To build a URI, we would follow the following guidelines.

1 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 the hierarchy and all that follow would be the next levels.

2 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 which in general language would be separated by a single space. From the API perspective, these words would collectively provide a meaningful concept.

3 Underscore ‘_’

Underscore should not be used in the URI design [9] because of a very common universally accepted practice of providing blue text colour and underlining font for a link, resulting in underscore making the link difficult to interpret.

4 Use of lower case letters only

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

5 Path variables

Parameters that serve a dual purpose: function parameters to the invoked controller method and a part of the URI of the resource, are the path variables. The resource identifier is the most common path variable as it serves a purpose in the identification of a resource in the controller as well as the URI of that resource. [10]

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 methods invoked, qualify as path parameters. These are mostly key-value parameters commonly used for queries. [10]

7 Hierarchy

The hierarchy of a URI is maintained by the use of ‘/’. Every resource will have a container class that holds multiple instances of the same resource type. This container is the first level of hierarchy for that resource type. 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, the hierarchy may also be applied with the use of ‘/’ as the designer deems fit. [1]

- **HTTP methods**

  Once we know the URI of a resource, we can perform a set of basic operations on it. As per the HTTP basics, the methods such as GET, PUT, POST, DELETE [11] can be applied while addressing a URI.

1 PUT [11] 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.

2 POST [11] 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.

3 DELETE [11] 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.

4 GET [11] 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.

2.1.8 HATEOAS [10]

Hypermedia as the Engine of Application State [10] refers to a web application framework by Spring [12] which provides the ability to build Java-based REST [1] applications with hypermedia [3] support.

A hypermedia web application, in contrast to traditional service-oriented architecture based web applications, will provide information on possible navigation options within the response. This eliminates the need to make requests to a staged specification whenever there is a need to make requests. HATEOAS [10] provides two main concepts that allow the application to have dynamic navigation possibilities.

- Relation

  Relation or simply *rel* [10] is the relation between the current resource that is embodied in the response and the resource that the link navigates to. The value of relation can either be a string value or ‘self’. In the case of a string value, the resulting URI of the linked resource would be the URI of the current resource suffixed with ‘/’ and the string value.

  In the case of ‘self’, this would imply that it is a self-referencing link and the URI of the resulting resource will be the same as current resource. [10]

- Link

  A link or *href* [10] is the embedded link within the response body of the resource. This is the absolute URL or the linked resource. The hashed combination of a relation and a link together represents a single *hyperlink.* [10] Hyperlinks are embedded within the body of the resource in the list named *links.* [10] This list holds all the hyperlinks for the resource.

  A JSON [6] representation is given below,

```
{
        "purchaseorder": {
          "startDate": "2014-12-31T22:00:00Z",
          "endDate": "2015-01-02T22:00:00Z",
          "links": [
                  {
                    "rel": "confirmation",
                    "href": "/purchaseorder/75648/confirmation"
                  }
          ]
        }
}
```

2.2 Overview of tools

2.2.1 Domain Specific language (DSL)

A DSL is a problem-oriented programming language that consists of abstractions and notations specific towards a solution to the problem rather than a generic form. [4] 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 the 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 (section 4), a detailed explanation would be given regarding the creation of a DSL.

2.2.2 Xtext [13]

Xtext is an Eclipse [14] based tool for creating DSLs. It allows you to create your own languages. It also provides the possibility to generate files from the provided information. Along with that it could also be useful for generating a UML [15] based model input because it supports other eclipse based tools like Ecore and Ecore model [16]. Xtext is extensively used in this research.

2.3 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.

2.3.1 Apiary IO [17]

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

2.3.2 Swagger [18]

This allows us to have a very neat and clean representation of our REST [1] application. It also allows us to test our API interface endpoints with a very friendly user interface. But it does not have a support for Spring's hypermedia framework [3] i.e. HATEOAS [10].

2.3.3 RAML [19]

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 [19] 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 HATEOAS [10], defining links between resources play a major role.

2.3.4 RestUnited [20]

RestUnited is another REST [1] API generator that allows the 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 [3] support in RestUnited.

2.3.5 Restlet Studio [21]

Restlet Studio is a REST [1] API generator with a web interface. It allows the user to create a REST [1] application and specify resources [1] along with hyperlinks [3]. This works well as it can generate the code we need. But it does not follow a model-driven approach [22]. 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. Thus, it can be seen that this does not serve the required purpose of reducing manual work. This is because the modelling is expected to be already achieved, and in the end, feeding information into a web interface is an extra work as such.

2.4 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 [10] framework to develop a REST [8] application. The behavioural aspects are covered by some of them but with different frameworks. As we focus on Spring HATEOAS [10] we are in need of something that supports it. The following is a list of existing technologies that have been studied.

Name	Pros	Cons
Apiary	• 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

Some of the technologies are really good for use with the structural aspects [2] of a REST application but they have limited support for behavioural aspects [23]. The following table shows the survey of the existing technology and their evaluation.

3 Research method

This section will highlight how the research was carried out and the procedure that was followed.

3.1 Scope

By knowing the language and framework that would be used, we can have a good idea on how the end code would look like. For that, we needed to clearly define the scope of the research. Using Spring boot application reference guide [24] 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+ [25]
- HATEOAS [10]
- Spring boot application and structure [24]

3.2 Steps

We have identified certain technologies that could help us in achieving the goals. Also, we have chosen some related work that could help us in defining a proper DSL. However, we have also analyses the potential drawbacks and shortcomings of the selected technologies and related work.

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

3.2.1 Information required from the user when creating a REST [1] application with HATEOAS.

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 is necessary to write the code manually in accordance with Spring boot application guidelines [24] 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. These files will be explained in detail with an example in Section 4.

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

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

Table 2. Files with information expected from the user

- **Name of the resource**
  The name of the resource is used in multiple places: the class declaration of all files, the names of all files and the root element name of resource.
- **Container mapping**
  The container mapping is the URI mapping declaration for the entire container. In HATEOAS, this is achieved by mapping the rest controller class with use of request mapping annotation. Thus, all the functions in the controller will be mapped relative to the mapping done for the controller.
- **Function mapping**
  The function mapping is the URI mapping declaration for every function in the rest controller. In HATEOAS, this is achieved by mapping each function with its own request mapping annotation. The annotation will have an attribute *value* which will identify this information.
- **Cardinality**
  Cardinality is required during generation of entity files by helping in declaration of the cardinality of all other entities that are contained by the subject entity.
- **HTTP method type**
  The HTTP method type, which is invoked on each state transition, is also required to generate hyperlinks. In HATEOAS, this is done by mapping each function with its own request mapping annotation which has an attribute *method.* This will store the information.
- **States of dynamic resources**
  The list of all the states of the dynamic resource is required in its controller functions and hyperlinks generation. This is achieved by creating an enumeration.

3.2.2 Syntax [2] to ensure that this information can be taken as input from the user while modelling.

Now that we have identified our inputs from the user, we need to come up with a syntax with the help of three chosen papers.

The study of the following papers has been conducted in order to come up with a suitable DSL to model a REST application. The objective was to study existing meta-models and to come up with a DSL which ensures that all required inputs are incorporated while modelling a Spring HATEOAS [10] application.

3.2.2.1 Modeling RESTful applications [2] by Silvia Schreier

This paper provides a very simple approach to Modelling REST 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, resource types, states and transitions, links and conditions etc.

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

The concept of *Action* in this paper is thorough. The paper has categorised various actions to suit different situations. Different *Actions* can be made on the resources, which would trigger a state change. This eventually led to the idea of providing many options in the *transitions* section of the DSL which will be described in detail later in the paper (section 4).

3.2.2.2 Towards a Model-Driven Process for Designing ReSTful Web Services [22] by Selonen et al

This paper provides a step by step approach to developing REST services. The paper addresses structural as well as behavioural modelling of an application. The example used in this paper is an airline booking system. The structural and behavioural 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 behavioural canonicalization model provides enough information about the application with respect to the addressees and data holders. In this paper, instead of using state chart model approach for behaviour modelling, the use of behavioural canonicalization is resorted to, 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. Here, addressees are resources along with accepted requests and bystanders are the information holders.

3.2.2.3 Modeling Behavioral RESTful Web Service Interfaces in UML [23] by Porres et al

This paper provides yet another approach to model-driven development where they address the structural as well as behavioural 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 a Conceptual model that has a collection and each collection has its element that can be addressed through the collection itself, maintaining hierarchy with its resource identifier. Hence, after the end of the conceptual modelling of an application, we end up having a system that serves hierarchy and follows the links to reach other resources. 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 serve as the URI. This idea would eventually be used in the code generation mechanism of this thesis, to reach the resources declared, from the root.

The behavioural model follows state chart based approach where the system is conceptualised as a bunch of states. Every state has a pre-condition and a post-condition. And then there is a transition with action and trigger. This concept of action and trigger has helped in this research in creating the concept of *transitions*, which would be explained further in section 4.

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

Required Info	**Selonen** [22]	**Schreirer** [2]	**Porres** [23]
Cardinality	yes	no	yes
Container Mapping values	yes	yes	yes
Function Mapping values	yes	yes	yes
Parameters	yes	yes	no
Transition Mapping values	no	no	yes
Function for transition	yes	yes	no

Table 3. Review of papers

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

3.2.3 A DSL that would take the identified inputs and would then generate the code.

In this step, we create a DSL using Xtext [13]. This DSL would conform to the syntax which we have formulated in the previous step. After that we would use the generator functionality of Xtext [13] to generate the end code.

This step must also ensure that the information taken from the 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 or 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 The RestGen language

RestGen is a very simple, elegant and easy domain-specific language that allows the end user to specify the structural and behavioural characteristics of an application and, in turn, generates a Spring HATEOAS [10] code from it. The generated code will be in Java.

As discussed earlier in this paper the scope of this work will be limited to Spring boot applications [24]. This section will help you understand the DSL more clearly. We will focus on a particular scenario viz. the equipment rental scenario [26], specifically handpicked to contain several situations of interest.

Every file that is to become a RestGen file is suffixed with a file extension of *.rg* after which it is well recognised as a RestGen file.

To understand how RestGen works, let us first understand the methodology that governs the idea of the language.

4.1 Methodology

The Institute of Computer Science at the University of Tartu conducts a course called Enterprise Systems Integration (ESI) [27] which deals with providing students with a good foundation to work and research on REST [1] services with the Spring framework [12]. Following concepts, which govern the overall methodology of the research, are based on a few concepts introduced during the course.

4.1.1 Scenario for Analysis

The following is a scenario obtained from the University of Tartu's web portal for the above-mentioned course [27]. It has been used exclusively in courses conducted at the University. The scenario chosen for this research is a subset of the bigger equipment rental scenario in which there are two systems. For the purpose of this paper, we will focus on only one of the systems.

4.1.1.1 The equipment rental Scenario [26]

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., to its customers via an online renting portal. Note, that in this domain, a piece of machinery is often referred to as a plant and, hence, we adopt this same convention.

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.

The *PurchaseOrder* contains the start and end dates of the rental period, 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 the decision with the reasons thereof, 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 the case of updating the purchase order, the new start date should have more than

18 hours of difference from the time of update to the time the plant has to be available as per the revised period of hire.

After the delivery is made, it can be accepted by the customer or rejected. In the case of rejection, the plant is brought back and based on the customer's remarks, the *PurchaseOrder* could be reopened and a fresh delivery made. Once a delivery is accepted 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 payment should be made. 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.

4.1.2 Analysis and design

The phase of analysis will undergo a 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 the identification process. For the analysis phase, we make use of some guidelines.

4.1.2.1 Resources and Entities

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 it is a resource but, it sure does qualify it to be one. If semantically the noun is found important then it can be considered as a resource. Nouns can be spotted differently in different languages. In the English language for example, 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 a potential *resource*. But semantically only PurchaseOrder can be a resource. Also *startDate* and *endDate* qualify as properties of a resource etc.

4.1.2.2 States and Transitions

The tense of the verb 'being' within the text usually helps us identify the state of a resource. Once we have identified the nouns that semantically qualify them as *resources* or *entities*, the tense of the verb 'being' that describe them can give us information about the state of the resource. Once again, semantically the tense of the verb should be important. If the verb is in its present tense i.e. 'is' then it would identify the current state of the resource in the context of the sentence. The future tense like 'become', 'will be', 'would be', or simply 'be' would identify the future state(s) of the resource from its current state.

E.g. in the scenario, 'the Invoice which is unpaid' implies that the current state of the invoice is *UNPAID* etc.

4.1.2.3 Relations

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

One must analyse the prepositions used to identify 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.

4.1.2.4 Actions

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.

4.1.3 Domain modelling

The classic domain model can be useful in distinguishing the *entities* from the *resources*. *Entities* most of the times end up also being *resources* but in many cases there are resources that may never be an *entity* in the 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 the lower-layer persistence of the application which plays a role in providing data to the higher level API. Usually, the domain model will begin with the data model of the application which consists of all the *entities.* Once refined, it will contain only the most important entities which will play an active role in providing data to the higher level API, and then it becomes a domain model. The entities which provide this data will remain while other will not be a part of the domain model.

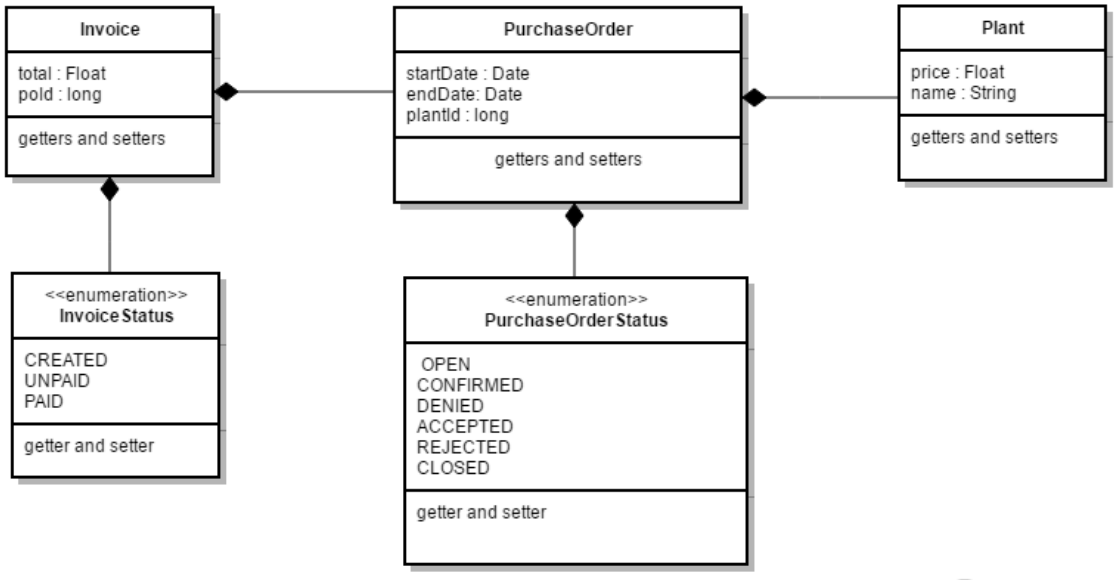

Figure 1. Domain model of Rentit

4.1.4 Resource modelling

Resource modelling is the next step towards designing a REST 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 though 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 *rel* (relation) in the URI.

Every container and resource combination will have basic Create, Read, Update, Destroy (CRUD) functions. There would be five functions as listed below:

Action	**HTTP method used**	**Invoked on**	**Mandatory parameters**
Read all resources	GET	Container	-
Create one resource	POST	Container	*body*
Read one resource	GET	Resource	*id*
Update one resource	PUT	Resource	*id, body*
Destroy one resource	DELETE	Resource	*id*

Table 4. Five basic CRUD functions

Create will be performed by a POST on the container. And Read all will be performed by a GET on the container. Read one will be performed by a GET on the item by use of a path variable in the URI being the identity of the single item. The update will be performed by a PUT on the item and Destroy will be performed by a DELETE on the item.

If a resource contains another resource, then it will be connected to that resource by a relation. This relation will be represented by a GET on the name of the resource in relation with the URI of the resource, e.g. /purchaseorder/{id}/plant, etc.

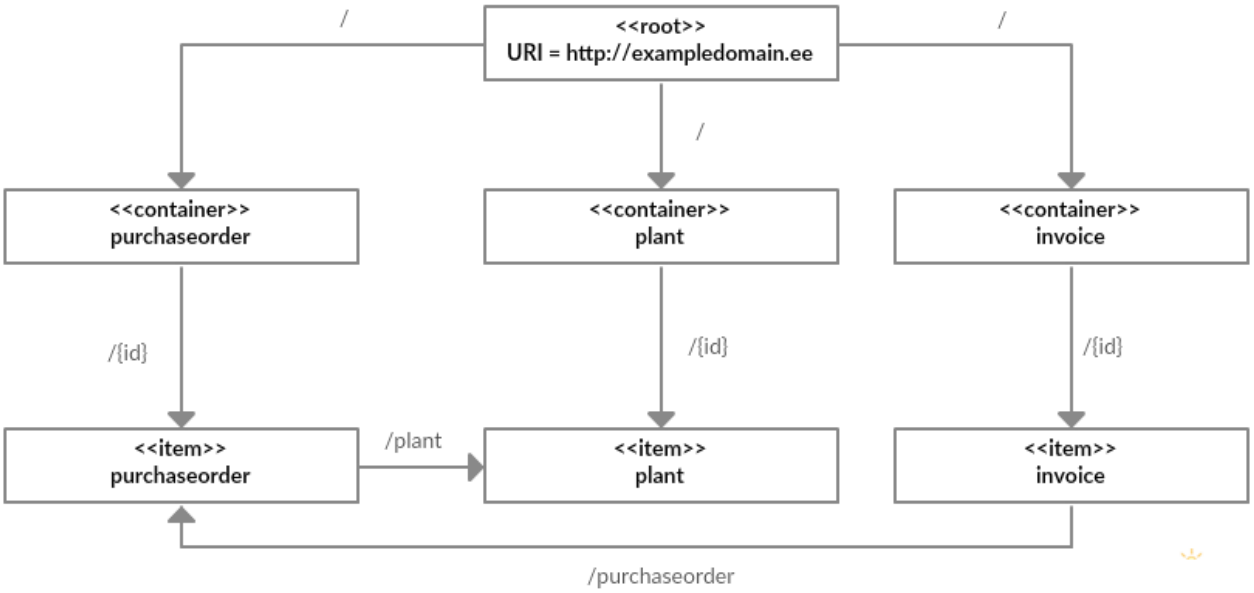

Figure 2. Resource model of Rentit

This leads to the following endpoints,

- /purchaseorder – GET and POST
- /purchaseorder/{id} – GET, PUT and DELETE
- /purchaseorder/{id}/plant - GET
- /plant – GET and POST
- /plant/{id} – GET, PUT and DELETE
- /invoice – GET and POST
- /invoice/{id} – GET, PUT, DELETE
- /invoice/{id}/purchaseorder – GET

4.1.5 Behavioural modelling (State diagram)

Key resources in the resource model will have multiple states. These are the resources whose behaviour changes as the application reaches the new stage in the life cycle. Every resource whose 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.

E.g. in the scenario [26], a *PurchaseOrder* has many states. Based on the information we get from the scenario, we can model a state diagram to represent the various states and the transitions between them. In ESI [27], the methodology includes the use of HTTP requests and the resource URI as triggers for each state transition. This makes it easier to understand the state transition in a RESTful way and helps in representing the actions.

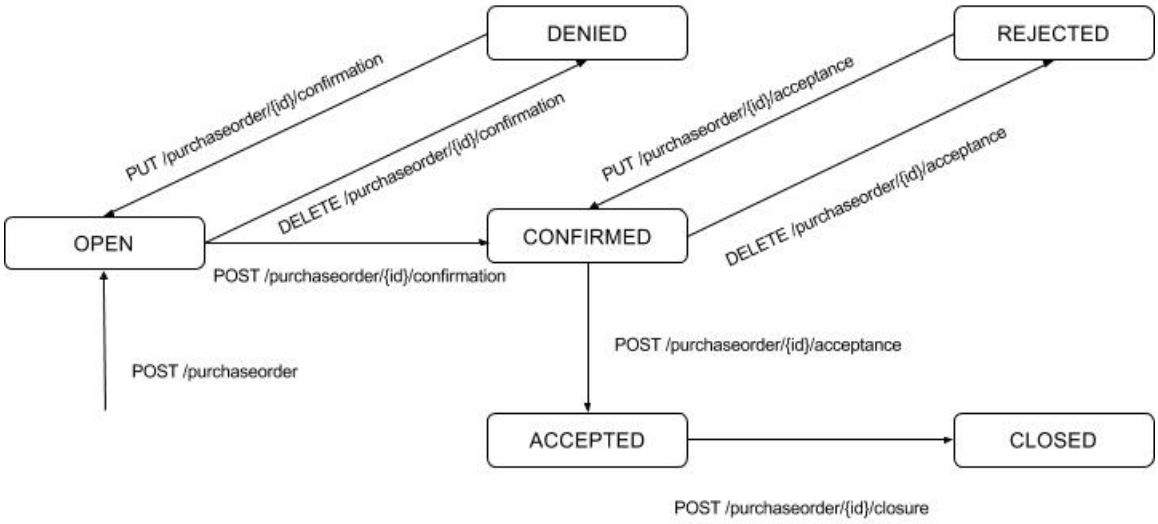

Figure 3. State diagram of a *PurchaseOrder*

4.2 RestGen Syntax

Now that we know the scenario, let us take a look at how it would cope with the proposed language, RestGen. In this section, we will address the DSL that we developed to gather information needed for building a REST application with Spring HATEOAS [10]. *XText* [13] provides a nice platform to create rules and logic for preparing a DSL. Making use of xtext's features we built an editor with a certain syntax. In order to avoid unnecessary user input, the syntax has been designed in a very simple manner. This section will help you understand the syntax. The editor will identify files with the extension *.rg* as a RestGen file.

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

The basic syntax of the *.rg* file is the following,

```
package somepackage
dbconf {
..
}
internal someinternalresource {
datatypes declarations …
  states {
  ..
  }
  transitions {
  ..
  }
}
More internals …
```

4.2.1 Terminology

For the language that is needed, this section will help enlighten some special terms that one should know before we could proceed. 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 equipment rental scenario [26] in section 4.1.1.1.

4.2.1.1 Package

A spring project like other Java [25] 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 [4]. A package could be declared by using the keyword *package* followed by the package string, such as the following,

```
package ee.ut.rentit
```

4.2.1.2 Database Configuration

Every project requires database configuration to be able to perform persistence operations. Spring HATEOAS [10] is no exception. Configuration of the database must be provided 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.

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

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

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

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

  All this information would be provided with the *dbconf* declaration. The above configurations would be embraced in curly braces after the *dbconf* keyword.

```
dbconf {
        database 'rentit-test2'
        username 'postgres'
        password 'letmein'
        host 'localhost'
        port 5432
}
```

4.2.1.3 State

A *state* is one particular user-defined state [10] of a particular resource. It is declared within the embracement of the *states* keyword and simply noted by its 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 are named in a capital snake-case fashion, but the user has the freedom to name it as preferred. In the end code, the name that is defined here will be used as it is.

4.2.1.4 States

States is a collection of multiple defined *state*. This collection can also be an empty collection. If not empty, it will qualify as an enumeration. We need this information because we need to know the different states that a particular resource can have. The embracement contains multiple *state* separated by a comma. It is defined using the keyword *states* and the information is wrapped around within two curly braces. It is defined within the declaration on an *internal* after the *datatypes*.

E.g. the states of *PurchaseOrder* from the scenario [26],

```
states {
	OPEN,
	APPROVED,
	DENIED,
	ACCEPT,
	REJECT,
	CLOSED
}
```

4.2.1.5 Transitions

A *transition* as the name suggests is a transition between two *state*. A *transition* will have numerous sub-entries to be filled by the user. A *transitions* declaration is a collection of multiple *transition.* It is defined within the declaration of an *internal* after the declaration of *states.* This collection is always empty, when the collection *states* is empty.

- **Trigger combination**

A transition will have a trigger in the form of a combination of an *HTTP method type* [28] and a relation to the current resource. A relationship or *rel* is a term that Spring [10] uses to define a self-referencing hyperlink. This means that a particular resource can have a link to itself but only after performing a minor change to itself. In this case, the change would be a 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 and DELETE [11] 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. It is identified as the *state* preceding the keyword *to.*

- **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. It is identified as the *state* succeeding 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 added 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 the form of a single-quoted string with the keyword *using*. The Syntax for transition will be as follows,

```
OPEN to APPROVED using 'approvePO' with POST on 'approval'
```

E.g. all transitions of *PurchaseOrder* from the scenario [26],

```
transitions {
	OPEN to APPROVED using 'approvePO' with POST on 'approval'
	OPEN to DENIED using 'denyPO' with DELETE on 'approval'
	APPROVED to ACCEPT using 'acceptPO' with POST on 'acceptance'
	APPROVED to REJECT using 'rejectPO' with DELETE on 'acceptance'
	ACCEPT to CLOSED using 'closePO' with DELETE on 'closure'
}
```

Here OPEN is a state and APPROVED is another state. When a resource is at state OPEN, it will have a relationship which would allow transition to another state which is APPROVED. This hyperlink accepts a POST request on the current resource and the logic for changing the state will be defined in ‘approvePO’ function of the controller. Same with OPEN to DENIED etc.

4.2.1.6 Internal resource (Internal)

An Internal resource is one which is specific to the domain of the application. In other words, this resource originates at the application and has an entity representation in the domain model. Every element of the resource model that is also a part of the domain model will qualify as an *internal*. E.g. from the scenario, a *PurchaseOrder* will qualify as an internal. Also, the container for the resource in the resource model does not need to be declared as an internal. E.g. declaration of *PurchaseOrder* in the *.rg* file will suffice.

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 equipment rental scenario [26] 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 resource identifier. We can make sure of this by putting the identity of the resource in the URI, hierarchically after the container and further creating a URI for each resource with the identifier on the path or the URI as a path variable [10]. With this, we can make sure that every *PurchaseOrderResource* will have a URI which will hold its identity and have a unique URI for itself. Also, the container for *PurchaseOrderResource* will have a URI for itself.

An internal resource can be declared using the keyword *internal* followed by other information which is enclosed in curly braces. Other information that has to be enclosed is explained below.

- Datatypes

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

  <javaType> <name>

  E.g. datatypes of *PurchaseOrder* from the scenario,

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

- 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. For example, from our equipment rental scenario, a *PurchaseOrder* will have a *Plant*.

  In this case, we need two things. First, 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 and ManyToOne.

  The syntax for an Internal Datatype is as follows,

  <cardinality> <internalDatatype> <name>

  E.g. internal datatype in *PurchaseOrder* from the scenario,

```
OneToOne Plant plant
```

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

E.g. the complete declaration of *PurchaseOrder* from the scenario,

```
internal PurchaseOrder {

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

  OneToOne Plant plant

  states {
       OPEN,
       APPROVED,
       DENIED,
       ACCEPT,
       REJECT,
       CLOSED
  }

  transitions {
       OPEN to APPROVED using 'approvePO' with POST on 'approval'
       OPEN to DENIED using 'denyPO' with DELETE on 'approval'
       APPROVED to ACCEPT using 'acceptPO' with POST on 'acceptance'
       APPROVED to REJECT using 'rejectPO' with DELETE on 'acceptance'
       ACCEPT to CLOSED using 'closePO' with DELETE on 'closure'
  }
}
```

4.2.1.7 External Resource (External)

An external resource is one that does not originate in the application domain but, either in the business logic of the application or in the 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 the case of the equipment rental scenario [26], 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, this resource would not have an entity entry in the database. Hence, there would be no need for creating entity specific files for such a resource. Every resource that is a part of the resource model but not a part of the domain model will qualify as an *external.*

In the equipment rental scenario [26] 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 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 [7] such as int, float, String etc. These will serve as the properties of this class and will determine the properties of the resource. The syntax is same as that of an internal resource's datatype.

  <javaType> <name>

  E.g.

```
java.lang.Float total
```

  The syntax for *external* consists of the keyword external followed by the data types embraced in curly braces separated by a line.

```
external PlantHireRequest {

	java.lang.String name
	java.lang.Float total
}
```

4.3 Mapping the information from User

Now that we know how to write the *.rg* file and the information we need to put into the same, let us take a look at how this information is used in the generation of the end code. The information from the *.rg* file is used in specific places (single or multiple places) in the end code.

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

The following files will be generated for every *internal* declared within our *.rg* file. For the sake of example let us consider the resource *PurchaseOrder* from our scenario.

4.3.1 Package and project structure

In our DSL, we provide an input called *package.* The information provided here comes to use when creating the package structure of the end code. The information provided here will serve as the base package for the project. The subsequent layers in the packaging structure will be relative to this base package.

In a Spring project, we observe that Spring creates some folders for us when we create a new spring project using *initializr*. The only folder that *RestGen* is concerned with or makes changes into is the *src/main/java* folder. RestGen will create all the packages in this folder. The first and foremost package that is created is the base package we discussed above.

The base package will contain two main files. The application file and the database configuration file. The application file contains the runner class for our application, which contains the *main* function used to run the Spring boot application. The database configuration file contains the standard JDBC setup.

From the information provided in the *package* declaration in our *.rg* file, we assume that the string value included as the last element in the package string is the name of the application, and will be used while naming these two files. This string value will be prefixed to *Application.java* and *DatabaseConfiguration.java* to create two of these files.

4.3.2 Application specific files

As per the observation, some files are application specific and need to be created only once. These files do not have any resource related information and lie at a higher level in the application. Some of these are the spring application files such as *Application.java* and *DatabaseConfiguration.java*, while some are Utility files like *ResourceSupport.java* and *ExtendedLink.java* etc.

The following are the files generated that come into the category of Application specific files. For the sake of example let us consider the files for a project with package declaration of *ee.ut.rentit*.

4.3.2.1 RentitApplication.java

This file contains the runner class of the application. The class contains the main function which runs the spring boot application.

4.3.2.2 RentitDatabaseConfiguration.java

This file contains the basic JDBC setup for the application. The class being annotated with *@Configuration* allows Spring framework to consider this as a configuration file. The information provided in the *dbconf* declaration in the *.rg* file will be used over here.

4.3.2.3 ResourceSupport.java

This is Spring framework's base class file that allows easy creation and manipulation of hyperlinks within a resource.

No information from the *.rg* file is required to generate this file.

4.3.2.4 ExtendedLink.java

This is an extension of Spring's *Link* class which allows the creation of links with traditional *href* and *rel*, as well as another entry called *method* which represents the method name in the controller function.

No information from the *.rg* file is required to generate this file.

4.3.3 Resource specific files

As per our observation, some file types are common to all resources in a Spring HATEOAS project. Every resource in the application will have one specific file of the type. Knowing the syntactical structure of a Java class file, the Spring style of coding and following the Java naming conventions, we have created patterns to map information provided in the *.rg* file to its target in the *.java* files for every *internal* that has been declared.

In this paper, we would provide the details using the resource *PurchaseOrder* from the above equipment rental scenario [26], and the files so generated.

4.3.3.1 PurchaseOrderResource.java

This file is stored in the *.rest* package relative to the base package. This file is the resource file in terms of Spring boot application [24]. This is a simple java class with an annotation *XMLRootElement* [12] with *name* attribute which gives it the name for XML [5] or JSON [6] 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 [10].

This class will be required for both, resources that originate at the application or *internal* as well as resources that do not originate at the application or *external*.

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 [5] and JSON [6] formats. Along with this, it would contain functions and data types along with their getters and setters. We would use the identity, *id* as a Long as per Spring boot guidelines. [24]

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.

  This information is available and made use of from the name of the *internal* in the DSL suffixed with 'Resource'.

- **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'.

  This information is available and made use of from the name of the *internal* in the DSL suffixed with 'Resource'.

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

  Every resource represented by a JSON [6] or XML [5] would have a root element name. This allows the recipient application to differentiate between different resources. We would name it the same as that of the class but in lower case. This is to follow the Spring naming conventions. [24]

  This information is available and made use of from the name of the *internal* in the DSL after converting it to lowercase letters.

```
@XmlRootElement(name="purchaseorder")
public class PurchaseOrderResource extends ResourceSupport{
```

- **Datatypes**

  Java [25] 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.

  This information is available and made use of from the *datatypes* declared in an *internal* in the DSL.

```
Date startDate;

public void setStartDate(Date startDate) {
	this.startDate =  startDate;
}

public Date getStartDate() {
	return startDate;
}

Date endDate;

public void setEndDate(Date endDate) {
	this.endDate =  endDate;
}

public Date getEndDate() {
	return endDate;
}
```

- **Embedded resources** (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 Equipment rental scenario [26] we observe that a *PurchaseOrder* contains *Plant.* Hence its resource will contain the resource of *Plant* etc.

  This information is available and made use of from the *internal datatypes* declared in an *internal* in the DSL.

```
PlantResource plantResource;
```

4.3.3.2 PurchaseOrder.java

This file will be stored in the *.models* package relative to the base package. This is the classic entity class in the model-view-controller world for a *PurchaseOrder*. This will have an annotation *@Entity* [12] 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* (section 4.2). This would be, same as the resource file, containing datatypes with getters and setters. Again, we would have an identity, *id* as a Long. This class will only be needed for the resources that are native to the application.

From the equipment rental scenario [26] 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 the 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 [12] class for the *internal* 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. This will make it easier to categorise the file and associate it with the resource.

  This information is available and made use of from the name of the *internal* in the DSL.

- **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.

  This information is available and made use of from the name of the *internal* in the DSL.

  E.g. for *PurchaseOrder* from the scenario,

```
@Entity
public class PurchaseOrder {
```

- **Datatypes**

  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 the entity, but to begin with, we will assume that the fields are same and later on it can be altered by the user as the software development cycle proceeds.

  This information is available and made use of from the *datatypes* declared in an *internal* in the DSL.

  E.g. for *PurchaseOrder* from the scenario,

```
	Date startDate;

	public void setStartDate(Date startDate) {
		this.startDate =  startDate;
	}

	public Date getStartDate() {
		return startDate;
	}
```

- **Entity Datatypes** (other Entities [12] of the application 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.

  E.g. in the Equipment rental scenario [26], we have a *PurchaseOrder* having one to one association with *Plant*. We can use the annotation *@OneToOne* before the declaration of the attribute *Plant*.

  This information is available and made use of from the *internal datatypes* declared in an *internal* in the DSL.

E.g. for *PurchaseOrder* from the DSL example,

```
@OneToOne
Plant plant;

public void setPlant(Plant plant) {
	this.plant =  plant;
}

public Plant getPlant() {
	return plant;
}
```

4.3.3.3 PurchaseOrderRestController.java

This will be the class that defines all the rest endpoints [10]. It will contain all the functions that are available as REST [1] services. This will be required by *internal* (section 4.3).

This class will have the annotation *@RestController* [10] and for every specific function, it may have the annotation *@RequestMapping* [10]. This annotation will have three mainly used attributes viz. *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 [5] or JSON [6] etc.

In this class, the functions usually have a direct access to the database with the use of repositories. Every function will perform some operation on the database. Hence, we need an *autowired* repository for the resource in question. The *@Autowired* [12] 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 generation of this class. According to the behaviour of the resource, different functions need to be placed here in this class to make a 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.

E.g. in the Equipment rental scenario [26] 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 numbers of functions in this class that would in turn serve as the hyperlinked [3] resource functions. These would have to be included 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 as such will also make it easier to locate.

  This information is available and made use of from the name of the *internal* in the DSL suffixed with 'RestController.java'.

- **Name of the class**

  To make things simpler, the name of the class will be thesame as that of the file. 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 subsequently in this section. Along with this, the class will also be mapped to a string value equal to lowercase of the name of the resource. This is done so by use of *@RequestMapping* annotation of Spring.

  This information is available and made use of from the name of the *internal* in the DSL suffixed with 'RestController'.

  E.g. for *PurchaseOrder* from the DSL example,

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

- **Name of the persistence repository**

  A repository in the Spring [12] world is an interface that has functionalities to perform persistence operations on the database. It is annotated with *@Repository* [12] and would also extend the *JPARepository* [12]. The *JPARepository* will have two attributes, first will be the entity [12] class and the other will be the data type of the identity field, which in 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 the functionality to perform such operations. So the declaration of the repository has an advantage. Injecting is done with the use of *@Autowired* [12] annotation before the declaration of the repository.

  This information is available and made use of from the name of the *internal* in the DSL suffixed with 'Repository'.

  E.g. for *PurchaseOrder* from the DSL example,

```
@Autowired
PurchaseOrderRepository purchaseOrderRepository;
```

- **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* [12] with attribute *method.* The value of the *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 an HTTP method type since there would be no default method type.

  This information is available to us and made use of with the *HTTPMethodType* in every *transition* of the *internal*.

- **URI mapping**

  Every function that is declared in the rest controller has to be mapped to a URI [10]. URI mapping can be achieved by declaring an annotation *@RequestMapping* [12] with the attribute *value.* The *value* can be a String value with '/' used to mark the beginning of a relative path or relationship.

  In Spring's [12] rest controller, a hierarchical approach towards mapping is followed. The first level of the hierarchy is the class and then the function. So the class itself can be annotated with *@RequestMapping* [12]. If so done, 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*, 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 a 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>'.

  This information is available to us and made use of from the *rel* in every *transition* of the *internal*.

- **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 an HTTP [1] call based on the trigger and method type. The transition functions will only be available in the controllers of those *internal* that have *states* with at least one *state* defined. The resources that do not have any states will not need to have these functions since there would be no transitions.

  This information is available and made use of from the *controllerFunction* property of every *transition* of the *internal*.

Transition functions are mostly same as regular functions in the declaration, i.e. they have an annotation *@RequestMapping* with attributes *method* corresponding to Spring's *RequestMethod* [10] 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 [12] interface.

E.g. for the following transition in *PurchaseOrder* from the DSL example,

```
APPROVED to ACCEPT using 'acceptPO' with POST on 'acceptance'
```

The corresponding entry in the controller would look like,

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

- **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 the modelling of robust and understandable REST [1] APIs. On the other hand, since they are merely conventions, the 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 in the above.

  These functions, similar to other functions in the controller, are annotated with *@RequestMapping* [10] with *method* and *value* attributes whose values correspond to Spring's *RequestMethod* [12] enumeration and relative path respectively. Some functions are mapped on the container while some are mapped on individual resources. 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 to 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 its type, which in the case of identity will be a Long along with an annotation *@pathvariable* [12].

According to Spring's conventions, every resource has five basic CRUD functions. RestGen will use these five basic CRUD functions for every resource.

1. GET on the container

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

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

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

   This will return the individual resource identified.

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

3. POST on the container

   This will create a new resource in the container.

```
@RequestMapping(method = POST)
public PurchaseOrderResource createPurchaseOrder() {
    PurchaseOrder purchaseOrder = new PurchaseOrder();
    purchaseOrderRepository.saveAndFlush(purchaseOrder);
    return purchaseOrderResourceAssembler.toResource(purchaseOrder);

}
```

4. PUT on an individual resource with identity as path variable

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

```
@RequestMapping(method = PUT, value = "{id}")
public ResponseEntity<PurchaseOrderResource> updatePurchaseOrder(
            @PathVariable Long id,
            @RequestBody PurchaseOrderResource
                        purchaseOrderResource) {
    PurchaseOrder purchaseOrder =
            purchaseOrderRepository.findOne(id);
    purchaseOrder = purchaseOrderResourceAssembler
            .fromResource(purchaseOrder, purchaseOrderResource);
    purchaseOrderRepository.saveAndFlush(purchaseOrder);
    return new ResponseEntity<PurchaseOrderResource>
                    (purchaseOrderResource, HttpStatus.OK);
}
```

5. DELETE on an individual resource

   This will delete the individual resource identified.

```
@RequestMapping(method = DELETE, value = "{id}")
public ResponseEntity<Void> deletePurchaseOrder(Long id) {
    purchaseOrderRepository.delete(id);
    return new ResponseEntity<Void>(HttpStatus.OK);
}
```

4.3.3.4 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* (section 4.2). This will be an attribute in the entity class of this resource.

Information required for this file is available and used from the *states* section of every *internal* in our DSL.

E.g. from the scenario, for a *PurchaseOrder*,

```
public enum PurchaseOrderStatus {
    OPEN,
    APPROVED,
    DENIED,
    ACCEPT,
    REJECT,
    CLOSED
}
```

4.3.3.5 PurchaseOrderResourceAssembler.java

This is a class that would extend *ResourceAssemblerSupport* of Spring [10]. The *ResourceAssemblerSupport* class provides a bridge between an entity and its resource. Once again this would mean that this class is not applicable to *external* since it would not have any entity to represent in the domain model.

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

This class would be responsible for adding hyperlinks [10] to a resource with the use of many convenient functions. Hence, the behavioural aspects [2] of a resource will play a vital role in the generation on this class. Based on the current state of a resource the hyperlinks [10] to be added will vary.

E.g. for the following transitions in the *.rg* file,

```
OPEN to APPROVED using 'approvePO' with POST on 'approval'
APPROVED to ACCEPT using 'acceptPO' with POST on 'acceptance'
```

The following entries will be made in the Resource assembler class of the corresponding *internal*,

```
switch (purchaseOrder.getPurchaseOrderStatus()) {
       case OPEN:
       purchaseOrderResource.add(
           new ExtendedLink(linkTo(methodOn(
                     PurchaseOrderRestController.class)
                          .approvePO(purchaseOrder.getId()))
                          .toString(), "approval", "POST"));
              break;
       case APPROVED:
       purchaseOrderResource.add(
           new ExtendedLink(linkTo(methodOn(
                     PurchaseOrderRestController.class)
                          .acceptPO(purchaseOrder.getId()))
                          .toString(), "acceptance", "POST"));
              break;
       case ACCEPT:
              break;
       case REJECT:
              break;
       case CLOSED:
              break;
       default:
         break;
}
```

In the equipment rental scenario [26] above, the PurchaseOrderResource could transit from state *OPEN* to either *APPROVED* or to *REJECTED*. Hence, at state *OPEN*, 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 [10].

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].

Information required for linking in this file is available and used from the *transitions* block of the *internal* in our DSL.

4.3.3.6 PurchaseOrderNotFoundException.java

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

The name of the *internal* is used to name this file.

5 Code generation

5.1 Parsing Input

In the earlier sections, we learnt about what information is required to generate the code in our chosen scope and framework. Also, we created a DSL [4] to take it as input from the user. We took a look at the DSL and we understood how to write a *.rg* file. This section will help understand the use of Xtext and Xtend that allowed the goal to be achieved.

5.1.1 About Xtext

Xtext is a framework which is based on eclipse [14], which provides a useful set of rules and definitions which, in turn, provide a good base for creating one's own grammar. With Xtext [13] one can accomplish parsing, linking and compiling. That, coupled with the possibility to export the product out as a plugin to the two most popular integrated development environments for Java: IntelliJ idea [29] and eclipse [14], 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 [13] DSL [4]. During the course of research carried out 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. However, in general, to be able to use Xtext, one may do any of the following:

- Installation eclipse afresh, 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 [4]. Create a new Xtext [13] 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 or UI support. In this paper, we will only address the DSL [4]. 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. 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 [4] was complete. In order to do so, some basics were needed to be understood.

5.1.3.1 Parser-Rules

The first two lines are the package and generator instance respectively. The main logic behind the DSL [4] starts after this. Every declaration in xtext [13] takes place in terms of *ParserRules* [35]. These are basic set of rules that define the every element 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 is contained in this rule.

5.1.3.2 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 listed 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 5. Native keywords of Xtext

5.1.3.3 Operators

Xtext [13] provides a variety of operators of which some had been used. The operators used in the DSL [4] 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 6. 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* from 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 created 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 an 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 the use of the syntax we formulated. 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, after generating the artefacts, having extension *.mwe2*, in the base package. This file, as mentioned before, 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 its *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 a *.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 needs 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 the generation process. There are two input parameters to this function. First, the instance of the *Resource* class which is a 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 subsections.

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 carried out in the section 4 i.e. the file contents required by a fully working Spring HATEOAS [24] project following the naming conventions. We already know what files are needed and their names. We could use the FSA instance to create these files and give them the content string with the use of its *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 the 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 it the file name along with its 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 each *.java* file could be divided 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 mentioned above, we would use the facility of function definitions within our generator to provide the contents of each file.

A function can be declared with the 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 triple quotes i.e. *'''*. What lies between these two *'''* and *'''* is what will be written into the file. [38]

Some files that need to be generated are application specific, thus will not hold any resource specific information. They will not hold any resource specific contents and hence will be created once. The resource specific files will have to be created 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 is 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 make a call to the functions we created for *internals* and *externals* respectively by making a simple check. The declarations for the creation of the non-resource specific files were made outside the loop and would be called only once.

A 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 which 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 loop 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 7. 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 file. 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 the use of *for* loops. With the use of *ImportManager* 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 the use of above mentioned dynamic content blocks. When the files are generated they are 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 any time. The generation merely creates an initial version for the user and the user can later edit the files or make changes to the *.rg* file.

5.2.2 Resource specific files

These files will be generated for all the *internal* resources declared in the DSL [4]. The following files would be generated for a resource called *PurchaseOrder* from the equipment rental Scenario [26]. With the use of *fullyQualifiedName* [39] function of Xtext [13], 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 certain places. The following are all the files that will be generated for an *internal e.*

5.2.2.1 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 [12]

  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.

```
(name="«e.name.toLowerCase»")
```

- The class declaration

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

```
public class «e.name»Resource extends ResourceSupport{
```

- JVM declaration

  For the JVM declaration, we would loop through the *datatypes* property of *Internal e*, which as per our DSL [4] is a list of all JVM [7] datatypes declared in that *internal.* Along with that, we must also declare getters and setters for the same.

```
«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.

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

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

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

5.2.2.2 PurchaseOrder.java

The resource specific information is gathered from the variable *Internal e* in the generator file.

- 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 the first letter capitalised.

```
@Entity
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 [4] along with their getters and setters.

```
«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 the spring application cardinality is denoted by declaring one of Spring's supported cardinality annotations. [12]

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

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

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

5.2.2.3 PurchaseOrderRepository.java

The resource specific information is gathered from the variable *Internal e* in the generator file.

- 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

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

5.2.2.4 PurchaseOrderRestController.java

The resource specific information is gathered from the variable *Internal e* in the generator file.

- 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.

```
@RestController
@RequestMapping(value = "«e.name.toLowerCase»")
public class «e.name»RestController {
```

- Repository injection declaration

  In usual practice, all persistence operations in spring application's controllers are performed by the repository injections. 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.

```
  @Autowired
  «e.name»Repository «e.name.toFirstLower»Repository;

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

- Class requests 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. [12]
  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.

  For example, the following is the generic function to create a GET function for a resource identified by its 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 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*.

```
«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»
```

5.2.2.5 PurchaseOrderResourceAssembler.java

The resource specific information is gathered from the variable *Internal e* in the generator file.

- 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. 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.

```
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 an HATEOAS project. The main objective of the function is to assemble the resource with all its properties and hyperlinks. Every time a resource is created it will have 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.

```
  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;
    }
```

In order to know 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.

5.2.2.6 PurchaseOrderNotFoundException.java

The resource specific information is gathered from the variable *Internal e* in the generator file.

- 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 the name of the resource is sufficient.

```
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 specific to the resource. Hence, they will not need *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 a 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,

1. **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 the *fullyQualifiedName* function. This file will be placed in the base package itself.

2. **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*.

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

3. **ExtendedLink.java** ***[27]***

This is a Spring provided file and it is used for providing useful functions to create hyperlinks in the resource assembler.

4. **ResourceSupport.java** ***[27]***

This is also a Spring provided file and is used for providing useful functions for resource building.

6 Conclusions

6.1 Work completed so far

6.1.1 Primary objectives

It can be concluded that the questions from the problem statement have been answered.

- We have managed to find out all the resource-specific information that is needed from the user for writing a Spring HATEOAS [10] application.
- We have managed to incorporate the structural and behavioural aspects [23] of a REST application into a DSL using Xtext along with an editor. [13]
- Further, managed to parse the information from it and generate code by creating a generator using Xtend. [38]

6.1.2 Collateral work

- We have managed to export the project as a plugin for all eclipse based IDEs.
- We have tested a Spring HATEOAS application with the equipment rental scenario [26] on Spring Tool Suite 3.7.2.RELEASE version.
- We have managed to make the plugin available as an Eclipse project update site [14] on *github*. [41]

6.2 Future work

In the future, there are plenty of things that could be done. Some of the things in our minds are the following:

- Dynamic database configuration supporting multiple databases.
- A plugin for the IntelliJ idea.
- Functionality for resource querying.
- Diagrammatic representation with the 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.

7 References

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

[2] S. Schreier, „Modeling RESTful applications," ACM, New York, 2011.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[22] P. Selonen, „Towards a Model-Driven Process for Designing ReSTful Web Services," IEEE, Los Angeles, 2009.

[23] Porres, „Modeling Behavioral RESTful Web Service Interfaces in UML," ACM, New York, 2011.

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

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

[26] P. Selonen, „Enterprise System Intergration,“ %1 *From Requirements to a RESTful Web Service: Engineering Content Oriented Web Services*, Berlin, Springer science+Business media, 2011, pp. 259-277.

[27] „Enterprise System Intergration Home page,“ 2014. [Võrgumaterjal]. Available: https://courses.cs.ut.ee/2014/esi/fall/Main/HomePage. [Kasutatud 18 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] „Github,“ [Võrgumaterjal]. Available: https://github.com/. [Kasutatud 18 May 2016].

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

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

[44] „Maven Jackson codehaus,“ [Võrgumaterjal]. Available: http://mvnrepository.com/artifact/org.codehaus.jackson/jackson-mapper-asl/1.9.13. [Kasutatud 18 May 2016].

[45] „Maven dbcp,“ [Võrgumaterjal]. Available: http://mvnrepository.com/artifact/commons-dbcp/commons-dbcp/1.4. [Kasutatud 18 May 2016].

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

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

Appendix

I. Installation

To install the application you must have installed an eclipse based IDE. The project is setup as a plugin on a github [41] page. The address to it is the following, [42]

http://vishalkirandesai.github.io

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 *RestGen* from the list and select Finish.

Creating a project

If you have installed STS then you can create a Spring boot app [24] from the IDE itself. If you have another eclipse based IDE then you would have to create a spring boot app from the spring initializr [43] 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
- JPA
- 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 [45]

```
<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 an extension, e.g. *rentit.rg* etc. This would be your main editor file for the RestGen DSL. When prompted by the IDE to select introduction of Xtext nature to the Project, select yes.

Start by declaring the *package*. The package should be the same as that in your Project. E.g. if the Project is *ee.ut.Rentit*, then the same should be declared in the *.rg* file as well. This will be the base package of the application. Then, declare the *dbconf*. After declaring *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.

You can start the application by running the project as a *Spring boot application*. [24] Note that all the mappings will be created, but you would still need to feed data into your database before you can make use of the application.

II. 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' dbName = STRING
	'username' username = STRING
	'password' password = STRING
	'host' host = STRING
	'port' port = NUMBER
;

terminal NUMBER:
	('0' .. '9')(NUMBER)*
;

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' '{'
			(states += 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

;
```

III. 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 IOutputConfigurationPro-
vider {

public final static String DEFAULT_OUTPUT_FINAL = "DEFAULT_OUTPUT_FINAL";

/**
* @return a set of {@link OutputConfiguration} available for the generator
*/
       public Set<OutputConfiguration> getOutputConfigurations() {
           OutputConfiguration defaultOutput =
             new OutputConfigurtion(IFileSystemAccess.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_OUTPUT_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);
       }

}
```

IV. Code for RgDslGenerator.java

```
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.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",
             "DEFAULT_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",
             "DEFAULT_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»Applica-
      tion.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://«rm.databaseConfiguration.host»:
                    «rm.databaseConfiguration.port»/«rm.databaseConfigura-
      tion.dbName»";

            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.get-
      Port() +                  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.toFirstUpper» «i.internal.name.toFirstLower»;

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

            public «i.internal.name.toFirstUpper» 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;
«FOR internal:e.internals»
import «internal.internal.fullyQualifiedName.skipLast(1)».models.«internal.inter-
	nal.name.toFirstUpper»;
import «internal.internal.fullyQualifiedName.skipLast(1)».rest.«internal.inter-
	nal.name.toFirstUpper»Resource;
import «internal.internal.fullyQualifiedName.skipLast(1)».rest.utils.«internal.in-
	ternal.name»ResourceAssembler;
«ENDFOR»
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
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.internal.name.toFirstUpper»ResourceAssembler
            «internal.internal.name.toFirstLower»ResourceAssembler =
          new «internal.internal.name.toFirstUpper»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»Reposi-
      tory.findAll();
        return «e.name.toFirstLower»ResourceAssembler.toResources(«e.name.toFirst-
      Lower»s);
    }

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

    @RequestMapping(method = POST)
    public «e.name.toFirstUpper»Resource create«e.name.toFirstUpper»() {
        «e.name.toFirstUpper» «e.name.toFirstLower» = new «e.name.toFirstUpper»();
         «e.name.toFirstLower» =
            «e.name.toFirstLower»ResourceAssembler.
                  fromResource(«e.name.toFirstLower»,
                        «e.name.toFirstLower»Resource);
        «e.name.toFirstLower»Repository.saveAndFlush(«e.name.toFirstLower»);
        return «e.name.toFirstLower»ResourceAssembler.
                  toResource(«e.name.toFirstLower»);
    }

    @RequestMapping(method = PUT, value = "{id}")
    public ResponseEntity<«e.name.toFirstUpper»Resource> up-
      date«e.name.toFirstUpper»(@PathVariable Long id, @RequestBody
      «e.name.toFirstUpper»Resource «e.name.toFirstLower»Resource) {
        «e.name.toFirstUpper» «e.name.toFirstLower» =
                  «e.name.toFirstLower»Repository.findOne(id);
        «e.name.toFirstLower» = «e.name.toFirstLower»ResourceAssembler.fromRe-
      source(«e.name.toFirstLower», «e.name.toFirstLower»Resource);
        «e.name.toFirstLower»Repository.saveAndFlush(«e.name.toFirstLower»);
        return new ResponseEntity<«e.name.toFirstUpper»Resource>(«e.name.toFirst-
      Lower»Resource, HttpStatus.OK);
    }
```

```
    @RequestMapping(method = DELETE, value = "{id}")
    public ResponseEntity<Void> delete«e.name.toFirstUpper»(Long id) {
        «e.name.toFirstLower»Repository.delete(id);
        return new ResponseEntity<Void>(HttpStatus.OK);
    }

        «FOR transition: e.transtions»
        @RequestMapping(method = «transition.methodType», value = "{id}/«transi-
        tion.rel»")
    public «e.name»Resource «transition.controllerMethod.toFirstLower»(Long id) {
          «e.name» «e.name.toFirstLower» = «e.name.toFirstLower»Reposi-
        tory.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.internal.name.toLower-
        Case»")
        public ResponseEntity<Void> modify«internal.internal.name.toFirstUpper»(
        @PathVariable Long id, @RequestBody «internal.internal.name.toFirstUpper»Re-
        source «internal.internal.name.toFirstLower»Resource) {
          «e.name» «e.name.toFirstLower» = «e.name.toFirstLower»Reposi-
        tory.findOne(id);
          «internal.internal.name.toFirstUpper» «internal.internal.name.toFirstLower»
        = «e.name.toFirstLower».get«internal.internal.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.internal.name.toLower-
        Case»")
        public ResponseEntity<«internal.internal.name.toFirstUpper»Resource> get«in-
        ternal.internal.name.toFirstUpper»(@PathVariable Long id) {
          «e.name» «e.name.toFirstLower» = «e.name.toFirstLower»Reposi-
        tory.findOne(id);
          «internal.internal.name.toFirstUpper» «internal.internal.name.toFirstLower»
        = «e.name.toFirstLower».get«internal.internal.name.toFirstUpper»();
              return new ResponseEntity<>(
              «internal.internal.name.toFirstLower»ResourceAssembler.toResource(
                    «internal.internal.name.toFirstLower»), HttpStatus.OK);
        }
        «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.toFirstUpper»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.toFirstUpper»Resource «i.internal.name.toFirstLower»Re-
      source;

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

            public «i.internal.name.toFirstUpper»Resource get«i.inter-
      nal.name.toFirstUpper»Resource() {
                  return «i.internal.name.toFirstLower»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 {
    «FOR state: e.states»
    «state.name»,
    «ENDFOR»
}
'''

// 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>{

      «FOR i:e.internals»
      «i.internal.name.toFirstUpper»ResourceAssembler
            «i.internal.name.toFirstLower»ResourceAssembler =
                  new «i.internal.name.toFirstUpper»ResourceAssembler();
      «ENDFOR»

    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 = cre-
      ateResourceWithId(«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»RestControl-
      ler.class)
                              .«transition.controllerMethod.toFirst-
      Lower»(«e.name.toFirstLower».getId()))
                              .toString(), "«transition.rel»", "
                                    «transition.methodType»"));
            «ENDIF»
```

```
            «ENDFOR»
                    break;
            «ENDFOR»
            default:
                break;
        }

        return «e.name.toFirstLower»Resource;
    }

    public «e.name.toFirstUpper» fromResource(«e.name.toFirstUpper»
      «e.name.toFirstLower», «e.name.toFirstUpper»Resource «e.name.toFirstLower»Re-
      source) {
            «FOR type:e.datatypes»
        «e.name.toFirstLower».set«type.name.toFirstUpper»(
            «e.name.toFirstLower»Resource.get«type.name.toFirstUpper»());
        «ENDFOR»
        «FOR i:e.internals»
        «e.name.toFirstLower».set«i.internal.name.toFirstUpper»(
            «i.internal.name.toFirstLower»ResourceAssembler.fromResource(
            «e.name.toFirstLower».get«i.internal.name.toFirstUpper»(),
            «e.name.toFirstLower»Resource.get«i.internal.name.toFirstUpper»Re-
      source()));
        «ENDFOR»
        return «e.name.toFirstLower»;
    }
}
'''

// 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));
    }
}
'''


}
```

V. Equipment rental scenario using RestGen (rentit.rg)

```
package ee.ut.rentit

dbconf {
	database 'rentit-test2'
	username 'postgres'
	password 'letmein'
	host 'localhost'
	port 5432
}

internal Plant {

	java.lang.String name
	java.lang.Float price

	states {

	}

	transitions {

	}
}

internal PurchaseOrder {


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

	OneToOne Plant plant

	states {
		OPEN,
		APPROVED,
		DENIED,
		ACCEPT,
		REJECT,
		CLOSED
	}

	transitions {
		OPEN to APPROVED using 'approvePO' with POST on 'approval'
		OPEN to DENIED using 'denyPO' with DELETE on 'approval'
		APPROVED to ACCEPT using 'acceptPO' with POST on 'acceptance'
		APPROVED to REJECT using 'rejectPO' with DELETE on 'acceptance'
		ACCEPT to CLOSED using 'closePO' with DELETE on 'closure'
	}
}
```

```
internal Invoice {

	java.lang.Float total

	states {
		OPEN,
		CLOSED,
		PAID
	}

	transitions {
		OPEN to CLOSED using 'close' with POST on 'closure'
		OPEN to PAID using 'pay' with POST on 'payment'
	}
}
```

VI. Generated Code for PurchaseOrderResource.java

```
package ee.ut.rentit.rest;

import java.util.Date;
import javax.xml.bind.annotation.XmlRootElement;
import ee.ut.rentit.rest.PlantResource;
import ee.ut.rentit.rest.utils.ResourceSupport;

@XmlRootElement(name="purchaseorder")
public class PurchaseOrderResource extends ResourceSupport{

PlantResource plantResource;

	public void setPlantResource(PlantResource plantResource) {
		this.plantResource =  plantResource;
	}

	public PlantResource getPlantResource() {
		return plantResource;
	}

	Date startDate;

	public void setStartDate(Date startDate) {
		this.startDate =  startDate;
	}

	public Date getStartDate() {
		return startDate;
	}

	Date endDate;

	public void setEndDate(Date endDate) {
		this.endDate =  endDate;
	}

	public Date getEndDate() {
		return endDate;
	}

}
```

VII. Generated Code for PurchaseOrderResourceAssembler.java

```
package ee.ut.rentit.rest.utils;

import ee.ut.rentit.models.PurchaseOrder;
import ee.ut.rentit.rest.PurchaseOrderResource;
import ee.ut.rentit.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>{

    PlantResourceAssembler plantResourceAssembler = new PlantResourceAssembler();

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

    @Override
    public PurchaseOrderResource toResource(PurchaseOrder purchaseOrder) {
        PurchaseOrderResource purchaseOrderResource =
          createResourceWithId(purchaseOrder.getId(), purchaseOrder);
        purchaseOrderResource.setStartDate(purchaseOrder.getStartDate());
        purchaseOrderResource.setEndDate(purchaseOrder.getEndDate());

        switch (purchaseOrder.getPurchaseOrderStatus()) {
          case OPEN:
          purchaseOrderResource.add(
                  new ExtendedLink(linkTo(methodOn(
                              PurchaseOrderRestController.class)
                                .approvePO(purchaseOrder.getId()))
                                .toString(), "approval", "POST"));
          purchaseOrderResource.add(
                  new ExtendedLink(linkTo(methodOn(
                              PurchaseOrderRestController.class)
                                .denyPO(purchaseOrder.getId()))
                                .toString(), "approval", "DELETE"));
                break;
          case APPROVED:
          purchaseOrderResource.add(
                  new ExtendedLink(linkTo(methodOn(
                              PurchaseOrderRestController.class)
                                .acceptPO(purchaseOrder.getId()))
                                .toString(), "acceptance", "POST"));
          purchaseOrderResource.add(
                  new ExtendedLink(linkTo(methodOn(
                              PurchaseOrderRestController.class)
                                .rejectPO(purchaseOrder.getId()))
                                .toString(), "acceptance", "DELETE"));
                break;
          case DENIED:
                break;
          case ACCEPT:
          purchaseOrderResource.add(
                  new ExtendedLink(linkTo(methodOn(
                              PurchaseOrderRestController.class)
```

```
                                .closePO(purchaseOrder.getId()))
                                .toString(), "closure", "DELETE"));
                        break;
                case REJECT:
                        break;
                case CLOSED:
                        break;
                default:
                        break;
                }

                return purchaseOrderResource;
        }

        public PurchaseOrder fromResource(PurchaseOrder purchaseOrder,
                        PurchaseOrderResource purchaseOrderResource) {
                purchaseOrder.setStartDate(purchaseOrderResource.getStartDate());
                purchaseOrder.setEndDate(purchaseOrderResource.getEndDate());
                purchaseOrder.setPlant(plantResourceAssembler.
                        fromResource(purchaseOrder.getPlant(),
                                purchaseOrderResource.getPlantResource()));
                return purchaseOrder;
        }
}
```

VIII. Generated code for PurchaseOrderRestController.java

```
package ee.ut.rentit.rest.controllers;

import ee.ut.rentit.models.PurchaseOrder;
import ee.ut.rentit.repositories.PurchaseOrderRepository;
import ee.ut.rentit.rest.PurchaseOrderResource;
import ee.ut.rentit.rest.utils.PurchaseOrderResourceAssembler;
import ee.ut.rentit.models.Plant;
import ee.ut.rentit.rest.PlantResource;
import ee.ut.rentit.rest.utils.PlantResourceAssembler;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

import static ee.ut.rentit.models.PurchaseOrderStatus.*;
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;

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

    @Autowired
    PurchaseOrderRepository purchaseOrderRepository;

       PlantResourceAssembler plantResourceAssembler =
             new PlantResourceAssembler();
    PurchaseOrderResourceAssembler purchaseOrderResourceAssembler =
             new PurchaseOrderResourceAssembler();

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

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

    @RequestMapping(method = POST)
    public PurchaseOrderResource createPurchaseOrder(
             @RequestBody PurchaseOrderResource
             purchaseOrderResource) {
        PurchaseOrder purchaseOrder = new PurchaseOrder();
```

```
    purchaseOrder = purchaseOrderResourceAssembler.
                fromResource(purchaseOrder, purchaseOrderResource);
    purchaseOrderRepository.saveAndFlush(purchaseOrder);
    return purchaseOrderResourceAssembler.toResource(purchaseOrder);
}

@RequestMapping(method = PUT, value = "{id}")
public ResponseEntity<PurchaseOrderResource> updatePurchaseOrder(
        @PathVariable Long id,
        @RequestBody PurchaseOrderResource purchaseOrderResource) {
    PurchaseOrder purchaseOrder = purchaseOrderRepository.findOne(id);
    purchaseOrder = purchaseOrderResourceAssembler.
        fromResource(purchaseOrder, purchaseOrderResource);
    purchaseOrderRepository.saveAndFlush(purchaseOrder);
    return new ResponseEntity<PurchaseOrderResource>(
        purchaseOrderResource, HttpStatus.OK);
}

@RequestMapping(method = DELETE, value = "{id}")
public ResponseEntity<Void> deletePurchaseOrder(Long id) {
    purchaseOrderRepository.delete(id);
    return new ResponseEntity<Void>(HttpStatus.OK);
}

  @RequestMapping(method = POST, value = "{id}/approval")
public PurchaseOrderResource approvePO(Long id) {
    PurchaseOrder purchaseOrder = purchaseOrderRepository.findOne(id);
    purchaseOrder.setPurchaseOrderStatus(APPROVED);
    purchaseOrderRepository.saveAndFlush(purchaseOrder);
    return purchaseOrderResourceAssembler.toResource(purchaseOrder);
}
  @RequestMapping(method = DELETE, value = "{id}/approval")
public PurchaseOrderResource denyPO(Long id) {
    PurchaseOrder purchaseOrder = purchaseOrderRepository.findOne(id);
    purchaseOrder.setPurchaseOrderStatus(DENIED);
    purchaseOrderRepository.saveAndFlush(purchaseOrder);
    return purchaseOrderResourceAssembler.toResource(purchaseOrder);
}
  @RequestMapping(method = POST, value = "{id}/acceptance")
public PurchaseOrderResource acceptPO(Long id) {
    PurchaseOrder purchaseOrder = purchaseOrderRepository.findOne(id);
    purchaseOrder.setPurchaseOrderStatus(ACCEPT);
    purchaseOrderRepository.saveAndFlush(purchaseOrder);
    return purchaseOrderResourceAssembler.toResource(purchaseOrder);
}
  @RequestMapping(method = DELETE, value = "{id}/acceptance")
public PurchaseOrderResource rejectPO(Long id) {
    PurchaseOrder purchaseOrder = purchaseOrderRepository.findOne(id);
    purchaseOrder.setPurchaseOrderStatus(REJECT);
    purchaseOrderRepository.saveAndFlush(purchaseOrder);
    return purchaseOrderResourceAssembler.toResource(purchaseOrder);
}
  @RequestMapping(method = DELETE, value = "{id}/closure")
public PurchaseOrderResource closePO(Long id) {
    PurchaseOrder purchaseOrder = purchaseOrderRepository.findOne(id);
    purchaseOrder.setPurchaseOrderStatus(CLOSED);
    purchaseOrderRepository.saveAndFlush(purchaseOrder);
    return purchaseOrderResourceAssembler.toResource(purchaseOrder);
}
```

```
        @RequestMapping(method = PUT, value = "{id}/plant")
        public ResponseEntity<Void> modifyPlant(
                @PathVariable Long id,
                @RequestBody PlantResource plantResource) {
          PurchaseOrder purchaseOrder = purchaseOrderRepository.findOne(id);
          Plant plant = purchaseOrder.getPlant();
                //TODO: write the modification lines.
                purchaseOrderRepository.saveAndFlush(purchaseOrder);
                return new ResponseEntity<Void>(HttpStatus.OK);
        }

        @RequestMapping(method = GET, value = "{id}/plant")
        public ResponseEntity<PlantResource> getPlant(@PathVariable Long id) {
          PurchaseOrder purchaseOrder = purchaseOrderRepository.findOne(id);
          Plant plant = purchaseOrder.getPlant();
                return new ResponseEntity<>(plantResourceAssem-
bler.toResource(plant),                HttpStatus.OK);
        }
}
```

IX. Generated code for PurchaseOrder.java

```
package ee.ut.rentit.models;

import java.util.Date;
import ee.ut.rentit.models.Plant;
import javax.persistence.OneToOne;
import javax.persistence.Entity;
import javax.persistence.Enumerated;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import static javax.persistence.EnumType.STRING;

@Entity
public class PurchaseOrder {

    @Id
    @GeneratedValue
    Long id;

    public Long getId() {
        return id;
    }

    @Enumerated(STRING)
    PurchaseOrderStatus purchaseOrderStatus;

    public PurchaseOrderStatus getPurchaseOrderStatus() {
        return purchaseOrderStatus;
    }

    public void setPurchaseOrderStatus(
            PurchaseOrderStatus purchaseOrderStatus) {
        this.purchaseOrderStatus = purchaseOrderStatus;
    }

    @OneToOne
    Plant plant;

    public void setPlant(Plant plant) {
                this.plant =  plant;
            }

            public Plant getPlant() {
                return plant;
            }

      Date startDate;

      public void setStartDate(Date startDate) {
            this.startDate =  startDate;
      }

      public Date getStartDate() {
            return startDate;
      }

      Date endDate;
```

```
	public void setEndDate(Date endDate) {
		this.endDate =  endDate;
	}

	public Date getEndDate() {
		return endDate;
	}

}
```

X. Generated code for PurchaseOrderNotFoundException.java

```
package ee.ut.rentit.rest.exceptions;

public class PurchaseOrderNotFoundException extends Exception {
    private static final long serialVersionUID = 1L;

    public PurchaseOrderNotFoundException(Long id) {
        super(String.format("PurchaseOrder not found! (PurchaseOrder id: %d)",
id));
    }
}
```

XI. Generated code for PurchaseOrderStatus.java

```
package ee.ut.rentit.models;

public enum PurchaseOrderStatus {
    OPEN,
    APPROVED,
    DENIED,
    ACCEPT,
    REJECT,
    CLOSED,
}
```

XII. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Vishal Desai**(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 the 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 the 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, **06.08.2016**