

UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Vinod Infant Dass John Rozario

Model-based Role Based Access Control for RESTful Spring applications

Master's Thesis (30 ECTS)

Supervisor(s): Dr. Luciano García-Bañuelos

Tartu 2016

Model-based Role Based Access Control for RESTful Spring applications

Abstract:

Model-driven software development is the modern software development methodology that aims at solving a specific problem by creating the domain models and providing the solution in a conceptual way. Domain-Specific Language (DSL) is the computer language that allows solving a problem in a specific domain. The goal of this thesis is to develop a software tool that helps to generate the software codes automatically with Role Based Access Control for a RESTful application. In this context, we want to provide a resource specification as an input to the software tool through DSL for describing the database layer components (i.e. Entity classes and Repositories), the integration layer components (i.e. Resources/Assemblers, Controllers), and a Role-Based Access Control model to be associated with a target application. Based on the definitions, our tools will generate code, including RBAC authentication/authorization related configuration and helper classes. Thus, the skeleton for the RESTful Spring Boot application with the annotations and basic code to enforce the RBAC model provided as input.

Keywords:

DSL, RBAC, REST, API, HTTP

CERCS:

P170 - Computer science, numerical analysis, systems, control

Mudelipõhine rollidel baseeruv ligipäässüsteem RESTful Spring rakendustele

Lühikokkuvõte:

Mudelipõhine tarkvaraarendus on kaasaegse tarkvara arendamise meetoodika, mille eesmärk on lahendada konkreetseid probleeme, luues domeeni mudelid ja pakkudes lahendust kontseptuaalsel viisil. Domeenipõhine keel (DSL) on arvuti keel, mis võimaldab lahendada probleeme konkreetses domeenis. Käesoleva lõputöö eesmärgiks on arendada tarkvaraline vahend, mis aitab luua automaatselt tarkvarakoodi rolli-põhise ligipääsu kontrolli abil RESTful rakendustele. Selle jaoks soovime pakkuda spetsifikatsiooni, mis läbi DSL-i sisendi kirjeldaks andmebaasi kihtide komponente (näiteks klassid ja hoidlad), vahekihtide komponente (st Resources/Assemblers, Controllers), ja rolli-põhise ligipääsu kontrolli mudelit koos sihtrakendusega. Tuginedes reeglitele, genereerib meie vahend koodi, mis sisaldab RBAC autentimisega / autoriseerimisega seotud konfiguratsiooni ja abiklasse. See on kogu RESTful Spring Boot rakenduse tuumik koos kommentaaride ning baaskoodiga, mille sisendiks on etteantud RBAC mudel.

Võtmesõnad:

DSL, RBAC, REST, API, HTTP

CERCS:

P170 – Arvutiteadus, arvustusmeetodid, süsteemis, juhtimine (automaatjuhtimisteooria)

Table of Contents

1	Introduction	6
1.1	Context	6
1.2	Motivation and problem description	6
1.3	Objectives	7
1.4	Thesis organization.....	7
2	Background	8
2.1	REST	8
2.1.1	Architectural constraints	8
2.1.2	HTTP methods	9
2.1.3	HTTP status codes.....	10
2.2	RESTifying approach	11
2.2.1	Domain model.....	11
2.2.2	Resource model.....	12
2.2.3	State model.....	13
2.3	Security.....	14
2.4	Role-based access control (RBAC).....	16
2.5	Domain Specific Language (DSL).....	17
2.6	Xtext and Xtend.....	18
3	State of the art	20
3.1	Related work.....	20
3.2	Discussion.....	21
3.3	Solution.....	22
4	Contribution	23
4.1	DSL specification	25
4.2	Code generation.....	30
4.2.1	From Domain model	30
4.2.2	From Resource model	32
4.2.3	From State model	34
4.2.4	From Role Based Access Control model	35
4.2.5	Views.....	39
4.3	Discussion.....	43
5	Case study	44
5.1	Example application (RentIt)	44
5.2	Summary.....	49

6	Conclusion and future work	50
6.1	Conclusion	50
6.2	Future work	50
7	References	51
	Appendix	53
I.	Grammar implementation	53
II.	DSL specification	55
III.	Project setup	57
IV.	Results of application – Views	59
V.	License.....	61

1 Introduction

1.1 Context

Web applications are ruling the internet world since the 1990s. In today's world, there are thousands of web applications. Web applications are usually implemented in programming languages such as Java, C#, PHP, etc. RESTful applications are becoming very popular since the early 2000s. RESTful web applications are used in almost all the industries such as banking, hospitals, government organizations, etc. For example, at first banking was a manual process, but now internet banking and mobile banking are replacing the manual banking process.

The RESTful services include CRUD operations such as create, read, update and delete. These operations make use of GET, POST, PUT and DELETE. Section 2 explains REST¹ in detail. RESTful APIs are one of the main elements for web applications. The RESTful services can be implemented on various platforms according to the requirement specifications. The RESTful application can also be cross-platform where one application is implemented in one programming language and another application in another programming language. In this case, both the applications can communicate with each other, share data and services through REST operations. REST API² is needed to make a REST call. Section 5 mentions the development of REST API. All the REST calls are public, but it can be restricted through security. There are many ways to secure the RESTful API, but one of the ways is through Role Based Access Control. Section 2.4 describes more about Role Based Access Control.

1.2 Motivation and problem description

REST web application had already made a significant impact in information technology. The interest shown towards the development of web APIs during my master studies and some interests on security services of web applications were the main motivation towards the selection of this thesis domain.

In today's world, developing a RESTful API from scratch is the big challenge for developers. It also takes several weeks and even months to develop APIs. Many companies use the process flow diagram or state diagrams to represent the entire process of their web applications. The writing code for the whole application takes long time and also needs more manpower. If the application has to be secured, then developers need to write code separately for the security configuration. Apart from back-end code, the developer has to write the front-end code as well. In the end, this thesis points on following research questions:

1. *Is there any way to generate the codes automatically for RESTful Spring Boot web application?*

¹ REST means Representational State Transfer which defines the software architecture style of the web application [1].

² API are the sets of requirements where one application communicate with another application [2].

2. *If possible, then how to generate the security configurations and views automatically to that web application?*

1.3 Objectives

The aim of this thesis is based on the achievement of the following contributions:

- Determine the appropriate security model for the RESTful API.
- Determine a way to generate the codes for RESTful API automatically.
- Develop a software tool to generate the RESTful API with appropriate security features.
- Generate the front-end codes for the RESTful application using our software tool.

1.4 Thesis organization

This thesis is organized as follows:

1. **Introduction** – introduces the general context of the topic, motivation and the problem of the thesis and the contributions to be achieved.
2. **Background** – gives an overview of REST and RESTifying approach that includes the domain model, resource model and REST API and also general overview about RBAC³.
3. **State of the art** – illustrates about the related work about this thesis, discussion about the related work and thesis and the overall solution to be implemented.
4. **Contribution** – illustrates about DSL specification and the code generation.
5. **Case study** – shows a case study and summary.
6. **Conclusion and future work** – concludes the work by making the overall overview of the thesis and the future work.

³ Role-Based Access Control

2 Background

In this chapter, the various technology backgrounds that are needed for this thesis will be discussed. The concepts that are to be discussed includes the RESTifying approach, RBAC, DSL, Xtext and Xtend.

2.1 REST

REST stands for Representational State Transfer, is one of the software architecture styles which was introduced and defined by Roy Fielding in 2000 [19]. REST architecture style is mainly based on the stateless, client-server and HTTP protocol. This architectural style is a key aspect in designing network applications and distributed systems. REST does not completely rely on HTTP but mostly linked with it. The properties of REST play the vital role in the REST architecture style and make the REST architecture simpler. REST architecture is a lightweight alternative to other mechanisms like RPC⁴, SOAP⁵, and WSDL⁶. [20]. Moreover; REST is a platform-independent, and language-independent service. The following sub-sections explain in detail about REST.

2.1.1 Architectural constraints

REST has a set of constraints to components, data elements, and connectors. The main constraints include client-server, stateless, layered system and uniform interface.

1. *Client-server:*

The client-server constraint is the most common constraint where the user-interface separates the clients from servers. Some properties or features are not mandatory for the clients, but some of those properties are important of servers. For example, some database related codes are not important to be present in client side, but it is more important for the server. Hence, portability of the code can be improved on the client side.

2. *Stateless :*

The client-server communication must be stateless in nature. The stateless nature is because that server must have all the information that are needed to respond to the request made by the client. The session state entirely depends on the client. In this constraint, the properties such as visibility, scalability and reliability are improved based on different aspects. The main drawback is that is the decrease in the performance of the network by sending the same data in the cluster of requests.

3. *Layered system:*

In REST architecture, a client cannot be able to tell how it has been connected to the end server either directly or through some midway. In the layered system of client-server connection, the client is connected to the server using client connector, client

⁴ Remote Procedure Calls

⁵ Simple Object Access Protocol

⁶ Web Services Description Language

cache, server connector, and server cache. These elements connect the client of the server, network or the database using the respective connectors.

4. *Uniform interface*

The visibility of the interactions is improved, and the system architecture can be simplified by applying the generality principle of software engineering [1]. REST interface is made to support large-gain hypermedia transfer. The uniform interface can be achieved by having multiple architectural constraints to guide the behaviour of components [1]. The four constraints of uniform interface are:

- Identification of resources
- Manipulation of resources through these representations
- Self-descriptive messages
- Hypermedia as an engine of application state.

These constraints are the main principles of REST. The resources know about the structure of URIs. For example, <http://www.anonymous.com/user/190> represents the dynamically pulled resource. The representations are mainly to transfer from JSON⁷ to XML⁸ and manipulate them. For example, the JSON snippet shown below describes the data to create an equipment.

```
{
  "name": "Excavator",
  "description": "2.5 ton excavator"
  "price": 250.0
}
```

This JSON is parsed in the server and saved to the database. The messages are mostly HTTP method which is explained in detail in next sub-section 2.1.2. The hypermedia indicates the stateless interactions where the server stores the data sent by the client.

2.1.2 HTTP methods

HTTP methods are used to map CRUD operations to HTTP requests [19]. HTTP methods are used with REST to form as a RESTful service. GET, POST, PUT, and DELETE are the four main HTTP methods.

1. *GET*

‘GET’ is used to retrieve the data from the database. The GET requests can be partial or conditional. The partial request retrieves all the information from the particular table. The conditional request retrieves only the specific data from the database based on the condition.

⁷ JavaScript Object Notation

⁸ Extensible Markup Language

Example:

GET /plant – gets all the plants of the respective table

GET /plants/1 – gets the plant with an ID of 1 of the respective table

2. POST

This HTTP method is mainly used to create a new entity in the table. However, it can also be used for update an existing entity.

Example:

POST /plant – creates a new plant

3. PUT

Like POST, ‘PUT’ can be used to create a new entity and also used to update an existing entity in the table. PUT is idempotent [19].

Example:

PUT /plant/1 – update the plant with an ID of 1

4. DELETE

If a resource has to be removed, then DELETE method can be used.

Example:

DELETE /plant/1 – deletes the plant with an ID of 1

2.1.3 HTTP status codes

HTTP response status codes are the codes that are results of the HTTP requests. When an HTTP request is made from the client, the server will send an appropriate status code along with the data, if any. The browser translates these status codes. The types of HTTP status codes are [19]:

- **1XX** - informational
- **2XX** - success
- **3XX** - redirection
- **4XX** – client error
- **5XX** – server error

2.2 RESTifying approach

The RESTifying approach includes the domain model, resource model, database, and REST API. Figure 2.1 explains the RESTifying approach. The following sections explain the artefacts shown in this approach. This approach is a key for creating the RESTful API.

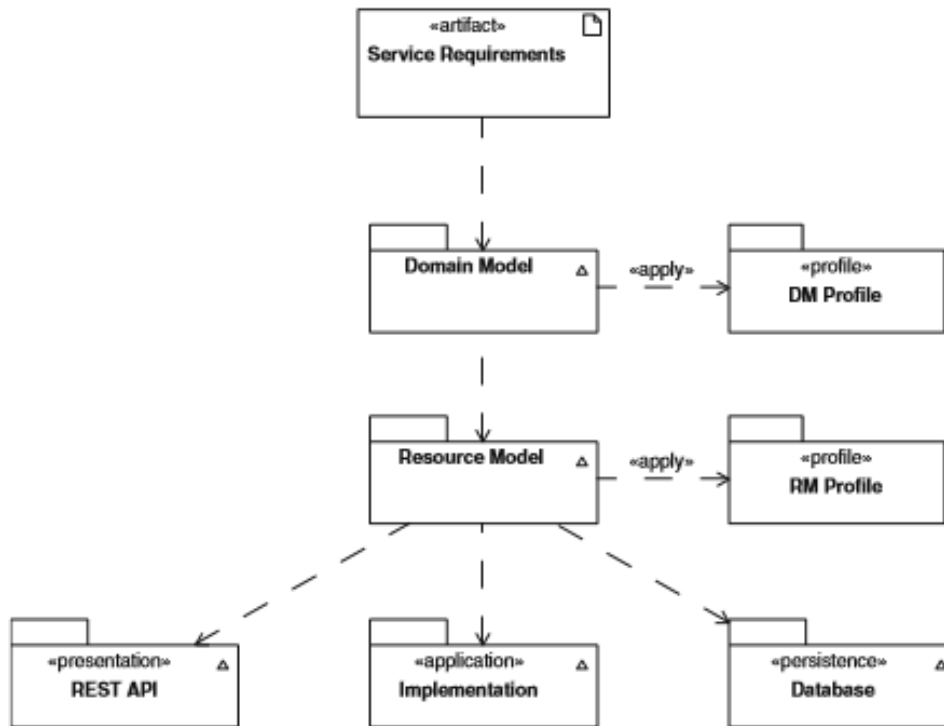


Figure. 2.1 RESTifying approach [3].

In this approach, domain model shows the data that is needed, the resource model gives the events and ‘behaviour model’ which is also known as state model that defines the states and the transitions of the RESTful API.

2.2.1 Domain model

Every application/software that we develop has the set of requirements to be fulfilled. The first step in the software development is creating a domain model. “Domain modelling (aka ontology modelling) is a method for describing the characteristics of and relationships between concepts in a specific domain or field of discourse” [4]. The service requirements are modelled as domain model. To define a domain model for the service requirements, some rules have to be followed. Associations can connect the communication between the elements of the domain model. The properties of associations include one to many, many to one, many to many and one of one relationship. As a way of example, Figure 2.2 shows a simplified domain model for a fictive web application from the equipment rental domain that is used in the context of the course Enterprise System Integration at University of Tartu [22].

The main element of this model is PurchaseOrder which has many equipments in it. The UML⁹ elements include classes, associations, compositions, generalizations, etc.

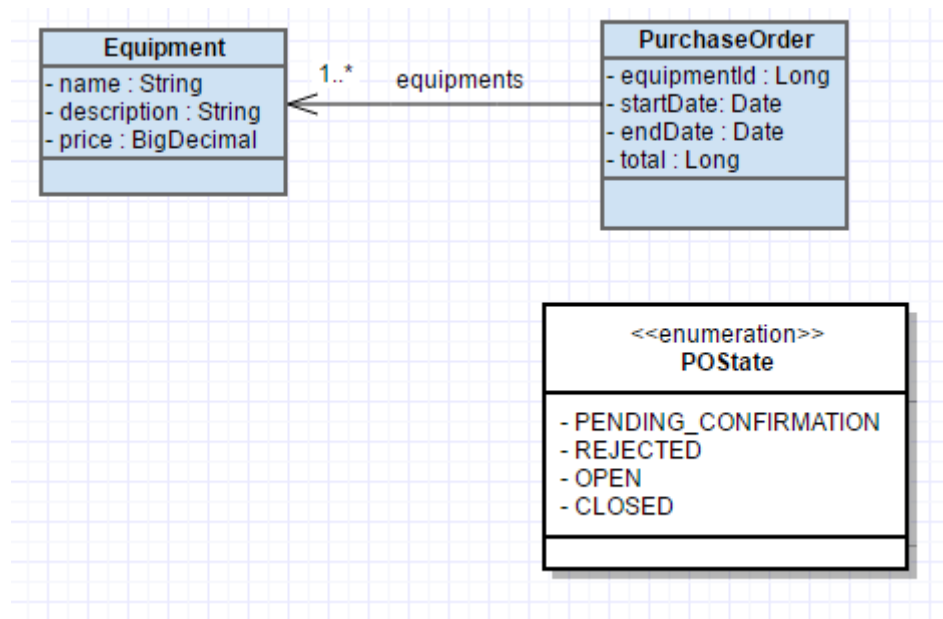


Figure. 2.2 Domain model of RentIt system

From the domain model, it is easier to define the elements RESTful API because the core elements of the RESTful API is the domain model. However, the resource model is needed to make the CRUD operations because the resource model connects with the database.

2.2.2 Resource model

The resource model is the modified version of the domain model which means that the domain model is re-organized to form the resource model. The resource model also defines the tables in the database. It also connects with REST API. All the data from and to the REST API is in the resource model of the application.

The resource model is modelled by using resource model profile. Figure 2.3 explains the resource model profile of RentIt system. The main elements of the resource model profile include Container, Item, Property, Item and Projection. It is possible to include HTTP¹⁰ operations and REST methods in the resource model. So, the RESTful web services can be defined fully from the resource model.

⁹ Unified Modeling Language

¹⁰ Hypertext Transfer Protocol

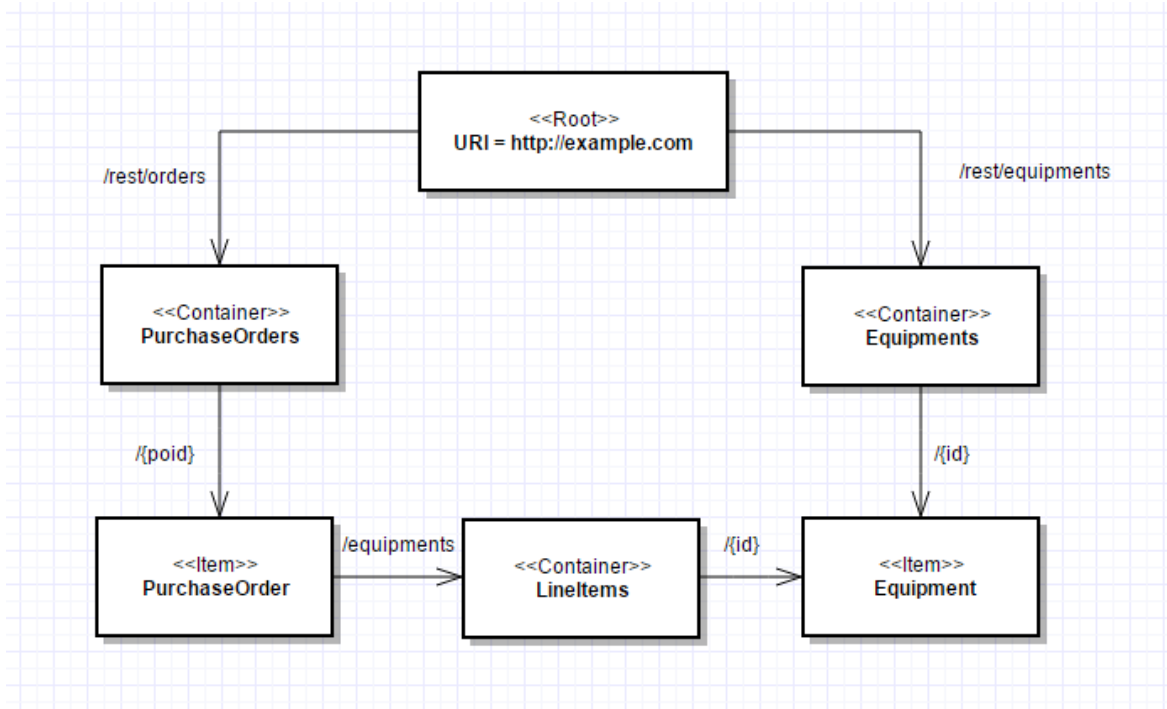


Figure. 2.3 Resource model of RentIt system

2.2.3 State model

The resource model provides the resources that are needed to build a RESTful API. However, RESTful API not only depends on the resources, but it also depends on the states and transitions of the application. So, the state model is needed to define the states and transitions of RESTful application. An example of state model is shown in Figure 2.4. The state model has the start state and the end state of the application.

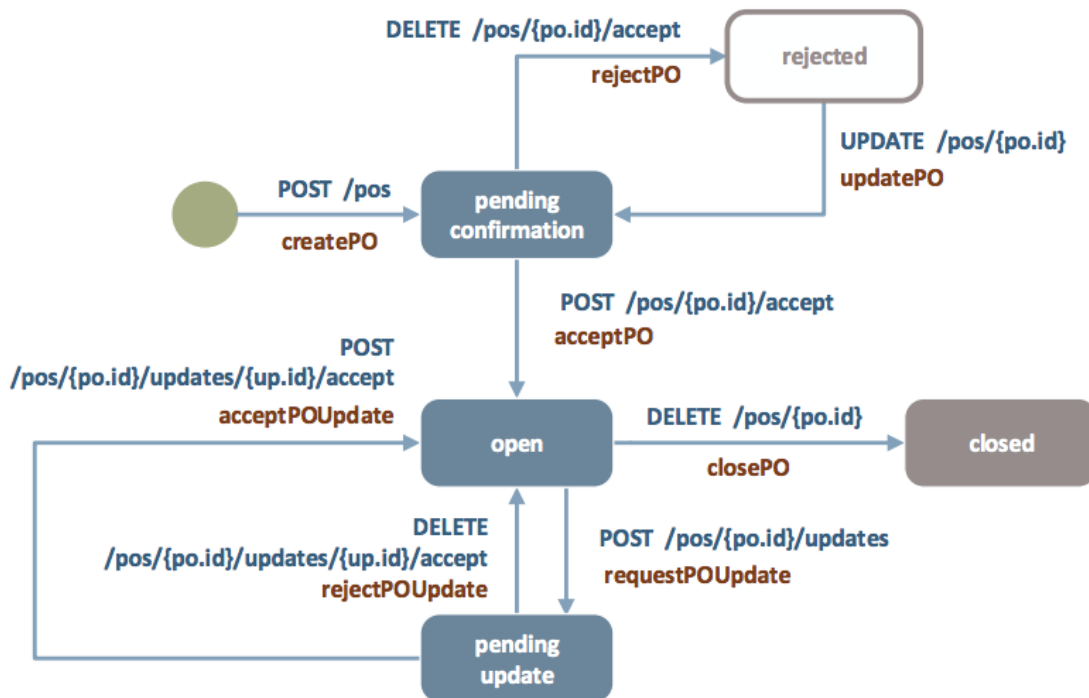


Figure. 2.4 State model of RentIt system [5]

So the REST application starts from the start state which travels to the end state. Each state has a name and the incoming and outgoing transitions. Each transition consists of HTTP verbs, URL¹¹, and a name. HTTP verbs will be any one of GET, POST, PUT or DELETE. The URL defines the path by which the specific transaction has to be accessed. In REST API, the transactions will be formed as methods which do the operations that have to be done in the particular state. By combing the resource model and state model forms the REST API. The resource model defines the resource/attributes of each transaction in the state model. Each transaction the state model has its properties in the resource model along with the transaction properties defined in the state model. All the states that are mentioned in the state model will be directed to do some operations in the RESTful API.

2.3 Security

One of the goals of this thesis is to implement security features in a RESTful API. The security is the way of protecting the system from an anonymous user access. In computer security, the data should be protected so that only authorized users can only access the data. This is typically known as *Information security*. *Authentication* and *authorization* are the two basic concepts in security. *Authentication* refers to the login access to the system. It means that the user who is registered to the system all with some role will have some login credentials to access the system. There are many ways to authenticate into system:

- Conventional username - password system
- Pattern based
- Fingerprint
- Image based
- Voice based password

These types of authentication depend on the user and the system. The system should have necessary configuration to provide any of these authentication types. *Authorization* is the next step of the authentication. The authenticated users will have access to only some parts of the system. This means of access to the part of the system is known as the system of authorization. In a network system, all the parts will have access to the administrator whereas some parts only are accessible to the other users.

For example, the study information system is a web application that have access to all the teachers and the students. However, the administrator will set the access rights to the students and the teacher separately so that the students cannot be able to access the teachers' private contents. There are many ways to implement security in the computer system. Different security models can implement the security in the system. Some of the models are listed and explained shortly below:

¹¹ Uniform Resource Locator

1. Access Control List (ACL):

Access Control List is a list of permission that is mapped to an object. It is possible to set the set of users to access the file in the system. All the entry or command that are specified in the list is in the form of subject and an operation. Role Based Access Control is a most alternative this model. The minimized RBAC model is equivalent with ACL group mechanism.

2. Bell-LaPadula model:

Bell-LaPadula model is a state-machine model which is used in the high-level security especially in government and military applications. This model is highly enforced access control model. The main features of this model include the Strong ★ property and Tranquillity principle. There are some strict limitations to this model.

3. Biba model:

Biba Model is also known as Biba Integrity Model. It is a formal state-transition system of security where there is a set of access control rules that are designed to ensure the data integrity. This type of security model is mainly for the high level of data integrity.

4. Brewer and Nash model:

This type of security was built to provide the security access control which can change dynamically. This is also known as Chinese wall model, which is constructed upon information flow model. The goal of this model is to provide the access controls that mitigate interest conflicts in organizations.

5. Graham-Denning model:

Graham-Denning model is defined as the model that illustrates how subjects and objects are created and deleted securely. It is mainly used in the distributed systems where it also shows how to assign access rights. This model is extended by Harrison-Ruzzo-Ullman security model where the security is based on the commands of respective conditions and operations.

6. Harrison-Ruzzo-Ullman:

This model is an extension of Graham-Denning model. This security is mainly built for the operating system where the model is for the integrity of access rights in the system. This model defines a security system that has set of universal rights and set of commands. The configuration is defined as a description and has a tuple of current subjects, current objects, and access matrix.

7. Lattice-based access control (LBAC):

Lattice-based access control model uses lattices to define the levels of security. It is a complex model whose access control is any combination of subjects and the objects.

It has a different type of security based on the partial order set. For example, if *user1* is only allowed to access *file-A* if the level of security of *user1* is greater than or equal to *file-A*.

8. Context-based access control (CBAC):

CBAC is mainly used in the network levels. It has a firewall software features that filters UDP and TCP packets that are in the application layer of session information. The benefits of CBAC includes:

- Real-time audit trails and alerts
- It can do the deep packet inspection
- Denial of service prevention and detection

9. Mandatory access control (MAC):

MAC security model is the high-level access control model for the operating system. It has a constraint that the ability of subject to perform some operation on an object. It is a strong security model that has many implementations.

10. Role-based access control (RBAC):

RBAC is a well-known security model that has been used by most of the enterprises that can implement MAC. This model focuses on the security model by which it restricts the access to the system based on the roles. It is the first alternative to Access Control List (ACL). The three primary rules for this model is:

- Role assignment
- Role authorization
- Permission authorization

With RBAC, it is possible to simulate LBAC. RBAC is widely accepted by many organizations and hence considered as a best practice security model.

All the above security models provide the security to the computer system. Most of the security models are applicable for operating systems and the network-side security. Among these models, ACL, Bell- LaPadula model, Biba model, LBAC and RBAC is suitable for securing web applications. However, Bell-LaPadula and Biba models are high-level security models which are complex to implement in web applications. RBAC is the superset security model for ACL and LBAC. Also, RBAC is widely used and has strong security features. So, RBAC will be a better selection for RESTful API. Role –Based access model will be explained more in detail in the next sub-section 2.4.

2.4 Role-based access control (RBAC)

A way of securing the system by restricting the access to the users of the system using authentication and authorization is called Role-Based Access Control. In RBAC, the system/application is secured by using the roles of the users. The RBAC security depends on

the types of role that the user has to access the system. The actions which are performed by the users are called transactions.

There are two types of access in RBAC. They are single role access control and multi-role access control. In the single role access, the system will always have only one role. The multi-role access system has a feature of registering a user to multiple roles. Figure 2.5 shows an example of multi-level RBAC. In the multi-level RBAC, the system is restricted to different roles of the system. Each role has access to an only specific set of transactions. The set of transactions are pre-defined in the system. When a new member is registered to the specific role, then the respective set of transactions are applied to that user. One of the significant uses of RBAC is that the security changes can be made easily.

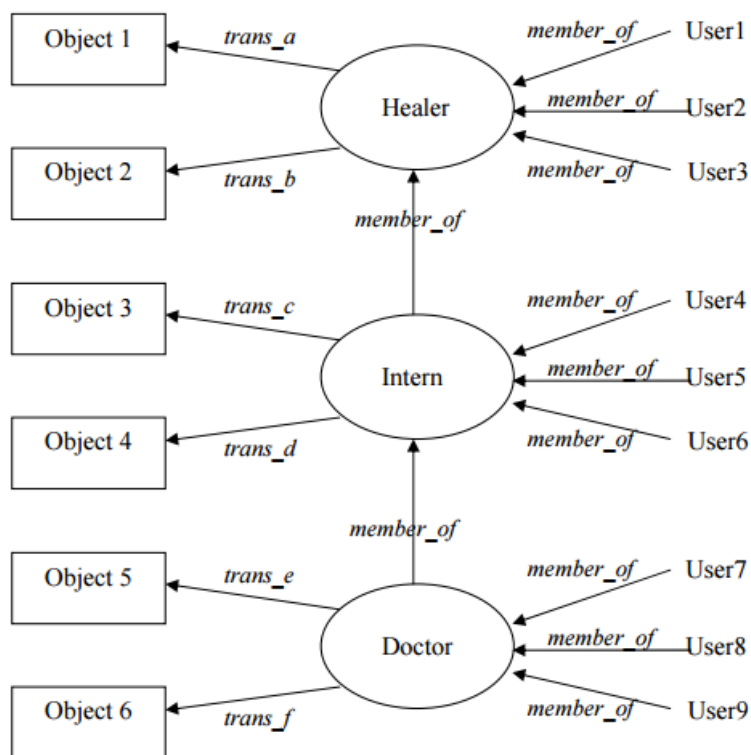


Figure. 2.5 Example of RBAC model [6].

2.5 Domain Specific Language (DSL)

Domain-Specific Language (DSL) is the computer language that allows solving a problem in a specific domain. “DSL is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on and usually restricted to, a particular problem domain” [15]. The main prerequisite for DSL is the thorough analysis of scenario and also should know about the structure of the domain of the application. It is better to have the structure of the application to be drawn as models so that it gives the clear idea to design a DSL.

There are many usage patterns for DSL [16]:

- It can be implemented by programming languages which can be later converted to host general programming language at run-time.
- The embedded form of DSL can be implemented as libraries
- Implementing DSL into the application either let the programmers to write the code manually or generate the code automatically. These two options can also be done together.
- Sometimes it can be used in command line for processing with the standalone tools by user operation.

For example, a part of DSL for state model shown in Figure 2.4 is illustrated below. The DSL shown below describes the states and transitions for creating, accepting, rejecting and deleting the purchase order.

```
states
  state initial
    create => pending_confirmation
  end
  state pending_confirmation
    acceptPO => open
    rejectPO => rejected
  end
  state rejected
    update => pending_confirmation
  end
  state open
    delete => closed
  end
  state closed
  end
end
```

2.6 Xtext and Xtend

A framework that is being used for this approach is “Xtext”. Xtext is a framework used for implementing programming languages or DSLs [13]. It is a plugin that can be added to the Eclipse framework. In our approach, Xtext is used to develop the DSL for our software tool. The Xtext project can be created as like creating the normal project in Eclipse. The files and packages generation in Xtext depends upon the type of the grammar. By the analysis of the types of grammar in Xtext, there are two main types: Common Terminals and XBase. XBase creates JVMmodel files in addition to the commonly generated files. For this approach, Common Terminals grammar type is used. Xtext has five main components. They are:

- Main
- Package that contains Xtext and Xtend files
- Generator
- Scoping
- Validation

Xtext generator generates files in separate folders. Parser and Serializer packages are also present in the Xtext project which parses the code and serializes them respectively. Xtext files contain the grammar that is needed for writing resource specification. Some rules should be followed to write the grammar in Xtext file. The Xtext grammar can be run as Xtext Artifacts.

Xtend is a high-level programming language especially made for Java programmers. It is a great advantage for Java developer. It is originated from Xtext, and no special plugins needed for it. It comes along with Xtext package. It is the main part of DSL development because it generates the codes for the RESTful API. The nature of the Xtend changes on Xtext grammar. If Xbase grammar model is chosen, then Xtend code should be in `JvmModelInferer`. This inferer extends from *AbstractmodelInferer*. This uses the Java Virtual Machine high-level programming.

3 State of the art

This chapter describes the research on the relevant papers and review about the papers. Also, the discussion about the related work regarding this thesis and the possible solutions to implement.

3.1 Related work

There are some relevant literatures that has been up to review to get the answers for the research questions mentioned in section 1.2. In 2006, Xin Jin did his master thesis work in “Applying Model Driven Architecture approach to Model Role Based System” [7] in which worked on a solution to build a tool-supported framework that uses the Model Driven Architecture approach with UML Class Diagram. The reason to review this literature is that this uses Model Driven Architecture to generate RBAC in XACML document. It shows how to create RBAC model. This paper focuses on generating security model in XACML¹² format automatically. The generated RBAC model is the class diagram which led the developer at the beginning of the software design process.

In 2007, Ahn Gail-Joon and Hu Hongxin did a distinguished work “Towards Realizing a Formal RBAC Model in Real Systems” [8] where the work aims at creating a framework called RAE (RBAC Authorization Environment). This framework is based on ArgoUML¹³ which is a UML-based modelling tool which converts the UML RBAC class diagram to generate Java code automatically. This tool also has specification and validation components to specify a more precise way of representing the data. The main idea behind the code generation is OCL¹⁴. The OCL library helps to generate OCL expressions to Java code. This literature depicts how Java code has been generated for RBAC model and also it shows a UML class diagram for RBAC model.

Similar to this paper, there is another research work “A UML Profile for Role-Based Access Control” [9] by Cagdas Cirit and Feza Buluca in 2009. In this paper, the author had proposed a UML profile for RBAC that gives an access control specifications to the whole development process at the beginning of integration process. In this paper, RBAC UML profiling is done based on its class diagram. UML profiling is done creating the stereotypes for all the elements of the UML that is to be created. The same UML stereotyping is also done for RBAC such as user stereotype, role stereotype, permission stereotype and few more security related stereotypes. The entire UML profiling is validated by using OCL.

In 2012, Manar H. Alafi, James R. Cordy and Thomas R. Dean did a research work on “Recovering role-based access control security models from dynamic web applications” [10] where they represented tool called ‘SecureUML.’ This paper uses PhpBB 2.0 as an example. This tool recovers RBAC model from the models of the dynamic web application. This article also has a future work for a large scale evaluation of the effectiveness test of

¹² Uniform Resource Locator

¹³ ArgoUML is an UML diagramming application written in Java

¹⁴ Object Constraint Language

this approach. In the mid of 2013, Kaarel Tark did his master thesis on “Role Based Access Model in XML-based Documents” [11]. In his work, the problem for securing XML documents has been solved. The solution is to implement RBAC in XML-based documents. In this approach, DSL used to implement RBAC security in the XML document. This paper also has the future work of applying the same method with improved XML schema.

When searching for latest model-driven development research papers, two papers were found. One of them is “Model-driven Development of RESTful APIs” [12] by Vitaliy Schreibmann and Peter Braun in 2015. This paper uses the DSL approach to generate the codes for REST API from the state model. However, this approach does not generate views and not deals with security. It uses the behaviour model that consists of states and transitions which in turn generate the respective code based on the states. Another paper is “Model-driven Testing of RESTful APIs” [13] by Tobias Fertig and Peter Braun in 2015. This article aims at developing the software generator to generate automated test cases. This software generator also creates the codes for REST API and test cases to test the REST API implementation. The testing types of this approach also include security testing.

3.2 Discussion

The literature review in section 3.1 gives some hints to make up the solution. The main literature to be considered is “Model-driven Development of RESTful APIs”. This is because that this paper has the DSL approach to generate REST API code from the state model. REST can be formally described by WSDL and WADL¹⁵ but not the RESTful APIs [12]. But replacing WADL with RDSL¹⁶ simplifies the definition of REST APIs [12]. So the overall architecture contains RDSL-API which is encapsulated by REST-API. Figure 3.1 below depicts the position of both REST and RDSL APIs.

Due to lack of support of security and API documentation in WADL, it has replaced by RDSL in this approach. There are few more considerations for RDSL. RDSL describes [14]:

- HTTP headers
- Resources
- URI parameters and templates
- Authentication mechanisms
- Methods that are linked with resources.

There are some topics like Media Types which are not focused in this approach. Another paper that can be considered is “Applying Model Driven Architecture approach to Model Role Based System”. Even though this approach deals with XACML, it has the implementation of RBAC with Model Driven Architecture. The another paper in “Towards Realizing a Formal RBAC Model in Real Systems” can be taken in to account because this approach is important for java codes and also implements RBAC.

¹⁵ Web Application Description Language

¹⁶ RESTful Service Description Language

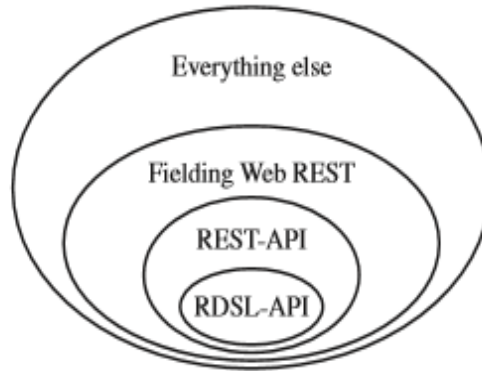


Figure. 3.1 The position of RDSL and REST APIs in relation to current API designs [12].

So, the three papers mentioned above will be considered to make the solution for the problem described in this thesis. The solution for this thesis problem will be explained in next section.

3.3 Solution

On deeper research about the papers mentioned in sub-section 3.2, using the implementations and methodologies can bring the solution to the problem of this thesis. The solution includes the development of a software generator in which state-resource model generates the skeleton code of RESTful API through DSL implementation. The developer then has to add some additional codes in order to make the full application to work more efficiently. The goal of this approach is to implement the RBAC to generated RESTful API. To achieve this, RBAC can be applied in DSL in which the Spring security authorization and authentication codes will be generated along with RESTful API. Also, to view the result of the RESTful API, the views are also generated through DSL. The user of this software generator has to specify the state model according to the rules of the grammar.

4 Contribution

The main scope of this thesis is the automatic code generation. In this section, we see how the code is generated by the software code generator from the resource specification. The subsections will provide details of how the code is being generated from the specification and setup generated files which are not specific of resource specification.

Let us consider an example of behaviour model shown in Figure 2.4. This is a model that corresponds to a fictive scenario inspired from the Equipment Rental domain (also known as the Plant Hire domain [22]), that is used in some courses at Software Engineering Programme of the University of Tartu. The scenario considers a fictive company, called RentIt, that provides access to its equipment catalog and allows its customers to place rental orders (a.k.a. purchase orders) via a web portal. Only a part of this model is taken for demonstrating this section. The simplified RentIt state model is shown in Figure 4.1. This RentIt state model illustrates the life cycle of a purchase order. It is basically an equipment rental system in which the customer chooses the equipment and creates purchase order.

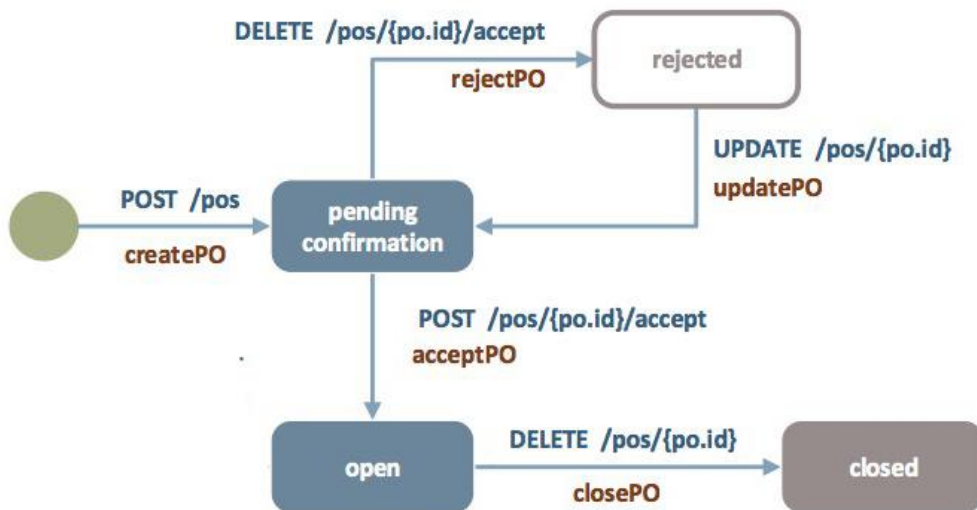


Figure 4.1 Simplified state model of RentIt system

Method name in Model	Verb	URI
createPO	POST	/pos
acceptPO	POST	/pos/{po.id}/accept
rejectPO	DELETE	/pos/{po.id}/accept
updatePO	PUT	/pos/{po.id}
closePO	DELETE	/pos/{po.id}

Table 4.1 Purchase Order methods

The purchase order will be created, accepted/rejected, updated or deleted in above model. Each operations shown in state model have HTTP method and REST path. Table 4.1 shows the method name, HTTP method and the path of the state model.

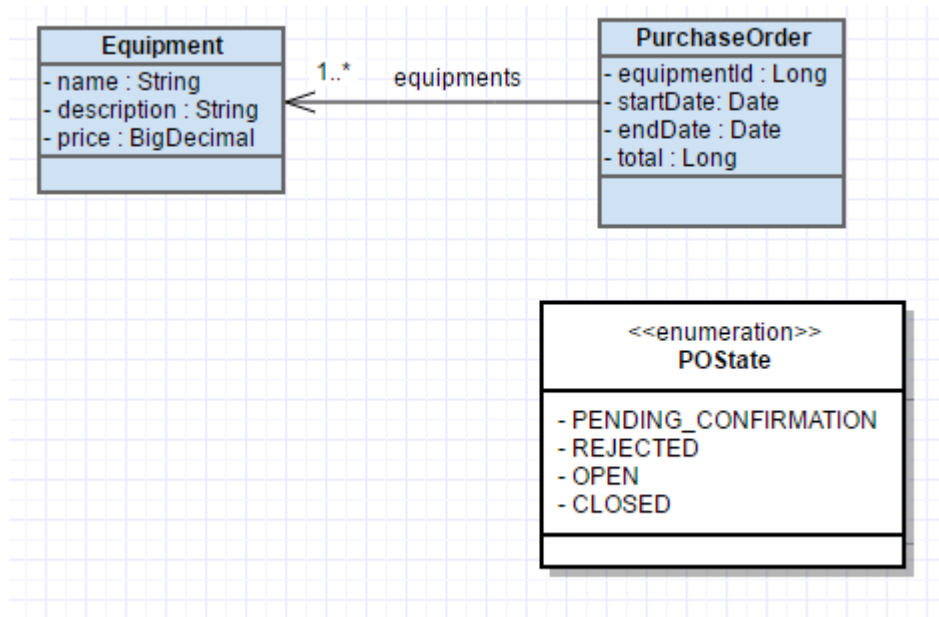


Figure 4.2 Domain model of RentIt system

The code generation for this model depend on the resource specification input to the code generator. The domain model and resource model of RentIt system is shown in Figure 4.2 and Figure 4.3 respectively.

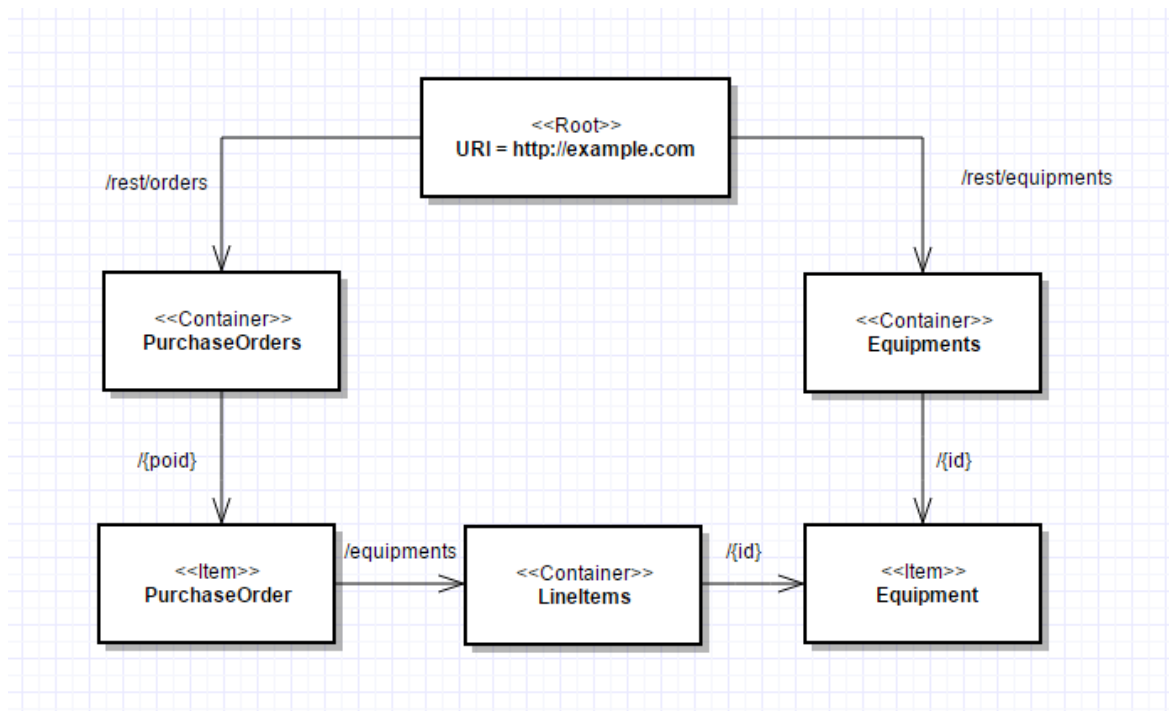


Figure 4.3 Resource model of RentIt system

4.1 DSL specification

One of the main parts of this thesis is resource specification. Resource specification is commonly defined as an input which is needed to generate the code. In other words, resource specification is the textual representation of data, resource and state model. Considering the state model shown in Figure 5.1, the specification of DSL is explained as follows.

From Figure 4.2, purchase order has one or more equipment(s). So, the resource specification will have two resources specified. The main part of this resource is ‘**resource**’ block. The resource is an *optional* element in the resource specification. It starts with a word “resource” and ends with “end”.

The resource block for Equipment is:

```
resource Equipment on "/rest/equipments" view "crD"  
end
```

Similarly, the resource block for purchase order is:

```
resource PurchaseOrder on "/rest/orders" view "CRUD"  
end
```

`"/rest/equipments"` is the root path of the Equipment controller and view `"crD"` defines the equipment has create, read and delete views. `"/rest/orders"` is the root path of the Equipment controller and view `"CRUD"` defines the equipment has create, read, update and delete views. Each resource declared has a name, path and view. The name of the resource is a word with only alphabets. The path string is *not optional*. If a resource does not have any root path, then it can be written as empty string. An important rule about path string is if a resource has a path string, then it should be preceded by “/rest”. The resource which has a word “rest” in its path will be secured. If a path does not have “/rest”, then it will be automatically added if:

- the path string is not empty
- path string does not have “/rest”
- Resource have view string or have at least an action.

View starts with word “view” and it is *optional*. Four characters “CRUD” which does the four operations Create, Read/List, Update and Delete respectively. The characters can be placed in the string at any index. This word/string is not case-sensitive. The HTML and JS files will be created based on the CRUD characters in the view string.

The data is an *optional* in the resource specification. The data starts with a prefix “data” and ends with “end”. All the data declared has name in left side and type in the right side with “:” in between. If the data is a list, then the word “many” can be used before data type. An important rule is if a resource has Create view then resource type has to be postfix by word “rendered” only if that data is one of the necessary element for creating the resource. From Figure 4.2, the data for the resources are written to the resource specification. The data for Equipment is:

```

data
  name: String
  description: String
  price: BigDecimal
end

```

The data for Purchase Order is:

```

data
  equipment : many Equipment rendered
  startDate : Date
  endDate : Date
  total : Long
end

```

In the above data, the data types which has been used should be declared all the above resources in the DSL. The datatype is an *optional* component in the resource specification. Whatever the data type that is being used in the resource block, it has to be defined as ‘datatype’. Since we have used String, BigDecimal, Date and Long in the above data, it has to be declared as data type along with void and Boolean.

```

datatype String
datatype void
datatype BigDecimal
datatype Date
datatype Long
datatype Boolean

```

The datatype which is written above should have the import of its respective package. All the data types declared except String, void, BigDecimal, Date and Long should be imported. The import is *optional* component in the resource specification.

```

import java.lang.Boolean

```

The ‘**actions**’ block is an *optional* component in the resource specification. It should start with prefix “actions” and ends with “end”. We shall specify actions for Equipment. The below action block shows create, update and delete methods of Equipment.

```

actions
  {"id" -> Equipment}

  create(Equipment) : Equipment
    with POST on ""

  update(Equipment) : Equipment
    with PUT on("/{id}"

  Delete() : Equipment
    with DELETE on("/{id}"
end

```

{"id" -> Equipment} specifies the id corresponds to the Equipment. The below action block shows the actions of Purchase Order. As we discussed in this section, the methods of Purchase order: create, update, accept, reject and delete (close).

```

actions
  {"poid" -> PurchaseOrder}

  create(PurchaseOrder) : PurchaseOrder
    with POST on "/pos"

  update(PurchaseOrder) : PurchaseOrder
    with PUT on "/pos/{poid}"

  delete() : PurchaseOrder
    with DELETE on "/pos/{poid}"

  acceptPO(PurchaseOrder) : PurchaseOrder
    with POST on "pos/{poid}/accept"
  rejectPO() : PurchaseOrder
    with DELETE on "pos/{poid}/accept"

end

```

{"poid" -> PurchaseOrder} specifies the id corresponds to the PurchaseOrder which is to be mentioned in parameter of the methods. The method names may not be same as in model because in our approach, the method names are CRUD should be same for all the mentioned resources. So, 'PO' is missing in CRUD operations. Since operations of delete method is similar to closePO, closePO is termed as delete. In the above case, if a resource has view string, then it will have some events automatically generated. Those are:

- Create – To create a respective resource
- Update – To updated the respective created resource
- Delete – To delete the created resource
- GetAll – To get all the data resource
- GetOne – To get one specific resource

All these automated events are given public access by default. An important rule is that the event name should not be duplicated. In case, if any of the above-mentioned events needed to change or if it is needed to include in states block, then it can be written in actions block. Only path string can be changed.

The states block is an *optional* component in the resource specification. This should start with prefix "states" and ends with "end". The states block can have any number of states. Each state starts with name with prefix "state" and ends with "end". Each state can have any number of transitions. Transitions are created with action name in the left side and another transaction name on the right side with implies (=>) in between them. The transactions can be created only with the events written inside "actions" block and with the existing states. Both state and transaction are *optional*.

From Figure 4.1, Equipment does not have any states. So, the Purchase Order states block is:

```

states
  state initial
    create => pending_confirmation
  end
  state pending_confirmation
    acceptPO => open
    rejectPO => rejected

```

```

    end
  state rejected
    update => pending_confirmation
  end
  state open
    delete => closed
  end
  state closed

end

end

```

Since ‘closed’ state does not have any transitions in the model, it is left empty without any transitions in the DSL.

The ‘role’ is an *optional* component in the resource specification. If any role is not declared or used, then the generated target application will have a default user as “admin”. If a role is specified in the specification, then the target application will have username and password as role name in lowercase letters in addition to admin user. In our example, we have two roles:

```

role Customer
role Works_Engineer

```

The roles are mainly used in methods of ‘actions’ block. If an event in an ‘actions’ block, has specific roles to be accessed, then it can be written inside [] preceded by word “roles”. If there are multiple roles for an event, then roles can be separated by comma. If the roles is not specified, then the respective event is accessed public. When the roles are added to the actions block of Equipment and PurchaseOrder, all the securities components are added to the respective resource. The modified Equipment actions block is shown below:

```

actions
  {"id" -> Equipment}

  create(Equipment) : Equipment
    with POST on "" roles [Works_Engineer]

  update(Equipment) : Equipment
    with PUT on("/{id}" roles [Works_Engineer]

  Delete() : Equipment
    with DELETE on("/{id}" roles [Works_Engineer]

end

```

‘roles’ mentioned above is optional. `roles [Works_Engineer]` is that all the operations can be done only by the works engineer not by any other roles except admin. Since our main focus is on the purchase order, the full description about the purchase order methods and its roles are shown in Table 4.2.

Method name in Model	Method name in DSL	Verb	URI	Roles
createPO	create	POST	/pos	Customer
acceptPO	acceptPO	POST	/pos/{poid}/accept	Customer, Works Engineer
rejectPO	rejectPO	DELETE	/pos/{poid}/accept	Works Engineer
updatePO	update	PUT	/pos/{poid}	Customer
closePO	delete	DELETE	/pos/{poid}	Customer, Works Engineer

Table 4.2 Purchase Order methods

As described in Table 4.2, the PurchaseOrder’s actions block has been modified as follows,

```

actions
  {"poid" -> PurchaseOrder}

  create(PurchaseOrder) : PurchaseOrder
    with POST on "/pos" roles [Customer]

  update(PurchaseOrder) : PurchaseOrder
    with PUT on "/pos/{poid}" roles [Customer]

  delete() : PurchaseOrder
    with DELETE on "/pos/{poid}" roles [Customer, Works_Engi-
neer]

  acceptPO(PurchaseOrder) : PurchaseOrder
    with POST on "pos/{poid}/accept" roles [Customer, Works_En-
gineer]
  rejectPO() : PurchaseOrder
    with DELETE on "pos/{poid}/accept" roles [Works_Engineer]

end

```

The ‘**package**’ is the first and *required* element in the resource specification. It starts with the word “package” which is followed by the package name of the target project.

```
package com.example
```

All the above mentioned specifications are the various parts of full resource specification. The complete resource specification is shown in Appendix-II. This example can be taken as account to write the new specification along with the respective state model. NB! Writing wrong resource specification without following the rules mentioned above and DSL will give the project run-time error which will not generate the code in the target project.

4.2 Code generation

In this section, the DSL specification which is described in the previous section is to be generated as code by the software generator tool.

4.2.1 From Domain model

Models describes the data elements of the RESTful API. The model is the main and basic component of MVC model which is foundation to the controller and assembler classes. Models is a package that is being generated automatically with a name “.models” with a prefix of package name. Models classes are generated from domain model. The Java classes inside this package are generated based on the resource specification. The purchase order resource specifications is,

```
data
    equipment : many Equipment rendered
    startDate : Date
    endDate : Date
    total : Long
end
```

and generated as model as shown below,

```
@Entity
@Data
public class PurchaseOrder {
    @Id
    @GeneratedValue
    Long id;

    @OneToMany(cascade = CascadeType.ALL)
    List<Equipment> equipment;

    @Temporal(TemporalType.DATE)
    Date startDate;

    @Temporal(TemporalType.DATE)
    Date endDate;

    Long total;
    @Enumerated(EnumType.STRING)
    PurchaseOrderState purchaseorderState;
}
```

In resource specification, the “**data**” block is generated as model java classes with respective resource name. Each model class has an ‘id’ variable generated along with other types. In the above purchase order model class, the id is annotated with Id and GeneratedValue. The model class is annotated by Entity and Data. Data is a part of Lombok that gives the getter and setters of the items that are presented in the model. In resource specification of purchase order, the equipment data has “many” items, so in the model class equipment is annotated by “OneToMany”.

Equipment model is also generated in a same way like as Purchase Order model. Purchase order resource contains “**states**” block in its specification, so an enum `purchaseorder-State` is generated. The state names are added to enum file `PurchaseOrderState` as shown below. These enum items will be added to the respective resource model class as with Enumerated annotation.

```
public enum PurchaseOrderState {

    INITIAL,
    PENDING_CONFIRMATION,
    REJECTED,
    OPEN,
    CLOSED;

}
```

Resources is a package that is being generated with a name “.resources” and a prefix of package name. Resources classes are generated from Domain model. The Java classes inside this package are generated based on the resource specification. As like Models, Resources are also generated from the “**data**” block of the resource specification with respective resource name and a postfix “Resource”. The number java classes created in models package will be same java classes created in resources package except enum files, if any. The resource class of purchase order is shown below,

```
@Getter
@Setter
@JsonInclude(Include.NON_NULL)
@JsonIgnoreProperties(ignoreUnknown = true)
public class PurchaseOrderResource extends ResourceSupport {

    Long idres;
    List<EquipmentResource> equipment;
    @Temporal(TemporalType.DATE)
    Date startDate;
    @Temporal(TemporalType.DATE)
    Date endDate;
    Long total;
    PurchaseOrderState purchaseorderState;

}
```

The data in the purchase order resource class is same as in models with the difference in the data annotations and id is replaced by ‘idres’. But the annotations for the resources are present with Lombok annotations- Getter and Setter and JSON annotations. All the classes in resource package are extends from “**ResourceSupport**” which is a helper class generated in utilities. Utilities is a package that is being generated automatically with a name “.utilities” and a prefix of package name. This package contains two Java files *ExtendedLink* and *ResourceSupport*. These files are the mandatory files that will be generated automatically and independent of resource specification.

The repositories are the important elements in MVC that connects with the respective database tables to makes the CRUD operations that are performed by the controller. Repositories is a package that is being generated with a name “.repositories” and a prefix of package name. Repository are interface classes that are created from “**resource**” name of resource specification with name as resource name with postfix “Repository”. The purchase order “**resource**” block is present in resource specification, so the purchase order repository is generated as,

```
@Repository
public interface PurchaseOrderRepository extends JpaRepository<PurchaseOrder, Long> {
}
```

All the repository interfaces in the repositories package extends from JpaRepository which has its respective model as its parameter.

4.2.2 From Resource model

The controllers are like communication component for the RESTful applications. It communicates between back-end and the front-end of the application. It controls the web application and makes the necessary operation that are needed for the application. ‘Controllers’ is a package that is being generated automatically with a name “.controllers” and a prefix of package name. The controllers package contains the controller java classes. The generation of controller java classes are depend on the resource specification.

If “**resource**” block is present in resource specification, then the respective controller will be generated. The generated resource controller java class has the name as “Controller” with prefix of resource name. All the generated controllers are annotated with RestController and RequestMapping. RequestMapping parameter is taken from path string (Eg: “/rest”) of the “**resource**” block. All the controller classes have declared respective repositories and assemblers. The purchase order controller is generated as,

```
@RestController
@RequestMapping("/rest/orders")
public class PurchaseOrderController {

    @Autowired
    PurchaseOrderRepository purchaseorderRepo;

    PurchaseOrderAssembler purchaseorderAssembler = new PurchaseOrderAssembler();

    EquipmentAssembler equipmentAssembler = new EquipmentAssembler();
}
```

The controller will have basic CRUD methods: create, update, delete, getAll, getOne. If anyone of these methods is found as event in actions, then the method will be replaced respectively. The purchase order controller has many methods in it. Purchase order event ‘accept’ method generation is shown in Figure 4.4. The other Purchase order methods such as rejectPO, delete, create, update are generated similarly to acceptPO.

Event in Resource Specification

```
accept(PurchaseOrder) : PurchaseOrder
  with POST on("/{id}/accept" roles [Customer, Works_Engineer]
```

generated as

Method in Controller

```
@Secured({ "ROLE_ADMIN", "ROLE_CUSTOMER", "ROLE_WORKS_ENGINEER" })
@RequestMapping(method = RequestMethod.POST, value =("/{id}/accept")
public ResponseEntity<PurchaseOrderResource> accept(PurchaseOrder purchaseorder, @PathVariable("id") Long id) {
    PurchaseOrder temp = new PurchaseOrder();
    temp.setPurchaseorderState(PurchaseOrderState.OPEN);
    PurchaseOrderResource purchaseorderResource = purchaseorderAssembler.toResource(temp);
    return new ResponseEntity<>(purchaseorderResource, HttpStatus.OK);
}
```

Figure 4.4 PurchaseOrder- accept method generation

The event name in the resource specification will be generated as method name. The input parameter type in specification will be generated as input parameter resource with RequestBody annotation. If the return type has “many”, then it will be generated as List of return type. If the path string contains any “id”, then those id are generated as PathVariable and placed as input parameter to the method. In all the controllers classes an exception handler method will be generated. It is annotated with ResponseStatus of NOT_FOUND and the ExceptionHandler annotation with the parameter of the respective handler classes which are generated separately in other package. Exceptions package contains exception java classes. Exception classes are generated with name ‘NotFoundException’ prefix of the respective resource name. These exception classes supports the respective controllers by handling the exceptions thrown by the methods in the controller during the run time of the application. Purchase Order exception is generated as,

```
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));
    }
}
```

The purchase order exception class is get called during the run time of the application when the purchase order is not found. Similarly, the Equipment exceptions class is also generated. The exceptions class always play a vital role in the controller.

4.2.3 From State model

Assemblers are the bridges for the controllers. It is mainly used for generating the links and for converting model to resource. Assemblers are generated as a package with name ‘.assemblers’ and prefix of package name. Assembler package contains assembler Java classes. These Java classes depend on resource specification. If a ‘**resource**’ block is present in the specification, then the respective resource assembler class will not be generated. All the assemblers that are generated for the resource models are extended from ResourceAssemblerSupport which is one of the libraries of the Spring Hateoas. All the resource assemblers will have only three methods defined in it. Those are two toResource() where one returns only one resource and other toResource() returns the list of resources. The third important method is checkRoles which check the roles of the method at the run time. A part of code generation of purchase order assembler is shown below,

```
public class PurchaseOrderAssembler extends ResourceAssemblerSupport<PurchaseOrder, PurchaseOrderResource> {
    EquipmentAssembler equipmentAssembler = new EquipmentAssembler();

    public PurchaseOrderAssembler() {
        super(PurchaseOrderController.class, PurchaseOrderResource.class);
    }
}
```

The function definition of toResource() depends on the resource specification. ‘**actions**’ events and ‘**states**’ transitions are used to create the links which are added to the created resource. The state model decides the process flow of RESTful service. The actions and states in the resource specification plays the main role in the assembler because it creates the links for the resource. There are two cases about toResource() will be discussed below:

1. *With State model :*

The assemblers works differently with and without state model. If the resource specification contains ‘**states**’ block with ‘**state**’ and transactions then, a switch block will be generated in toResource method. All the state names will be generated as switch cases, each transaction will be formed as a switch case definition. Every transaction that is being defined inside a case definition, has an if block where it checks the event roles with a present role using checkRoles() during the run time of the application.

If there is ‘Initial’ or ‘start’ state in resource specification, it will be ignored in the code generation and will not be generated as switch cases. Let us consider the actions and state blocks of purchase order. There are many events and states in the purchase order. The update purchase order and Pending state is taken as an example for code generation Figure 4.5 shows how the switch case is generated for update purchase order and Pending state from the resource specification. Similarly, the other purchase order states such as rejected, closed, open are also generated as switch cases and the respective transitions are generated as switch case definitions.

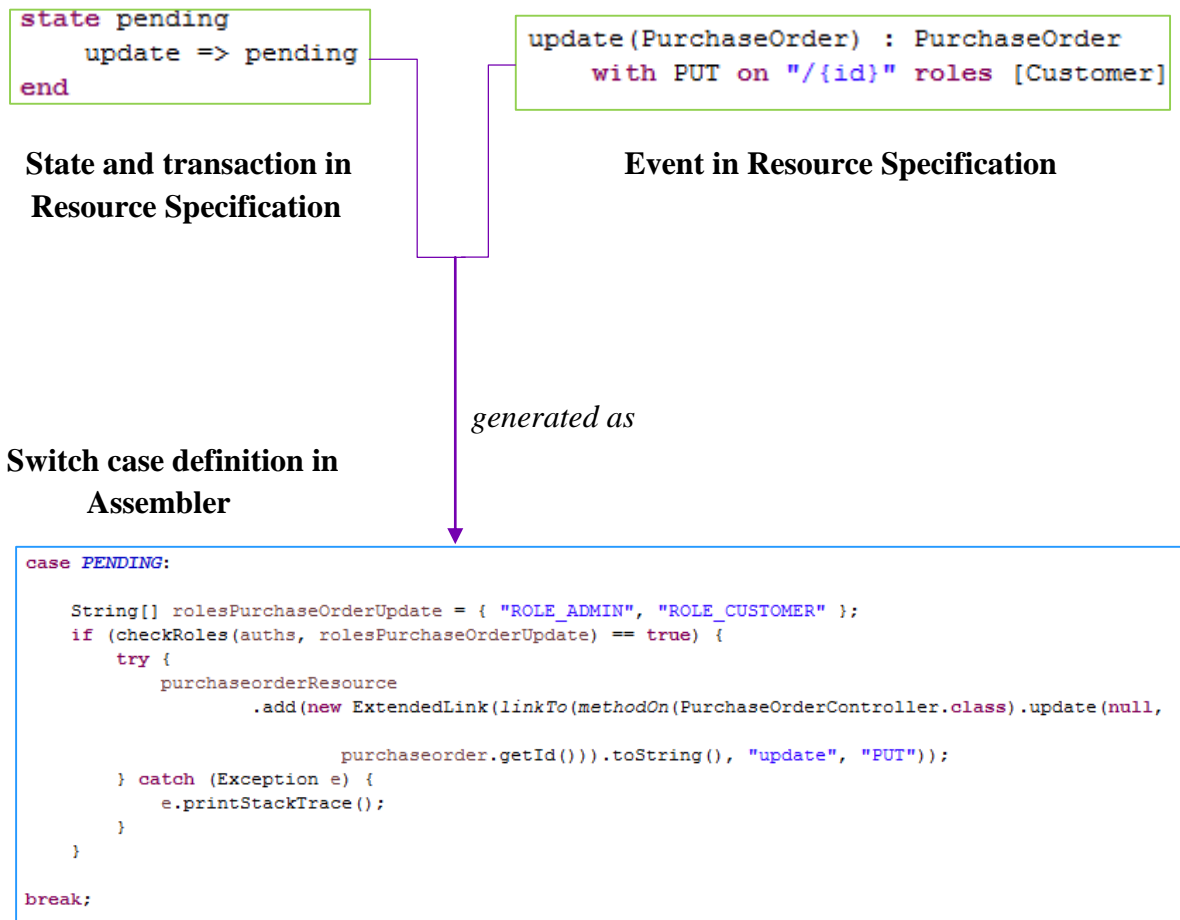


Figure 4.5 Update () and Pending state code generation.

2. Without State Model :

The state model is optional in resource specification and code generation. Even without the state model, the assemblers can be generated with the exclusion of switch cases. But the extended links will be generated. The link generation depends on the 'view' string or events in 'actions' of resource specification. In resource specification, if CRUD methods for a resource is present in actions block, then the links for those events are generated with roles check, if specific event has roles. If the CRUD method is not present in the resource specification, then the links will be generated without roles check block.

4.2.4 From Role Based Access Control model

The authentication and authorization through RBAC is the main goal of this approach. The security classes are generated from RBAC security model. The 'role' in the resource specification describes the roles of the target application. The roles which are declared is used as 'roles' as a list in the events of actions block. If some roles is assigned to an event in resource specification, only those roles can be able to access that specific event. For other roles, access is restricted to that event.

There are two model classes: *Users* and *Authorities* which are generated automatically independent of resource specification. Users is a class for new user registration that has a username (primary key), password and enabled. Authorities class is a model for assigning a role (authority) for the user has an id (primary key), username and authority.

```
public class Users {
    @Id
    String username;
    String password;
    boolean enabled;
}

public class Authorities {

    @Id
    @GeneratedValue
    Long id;

    String username;
    String authority;
}
```

There are also other two Java resource classes *AuthoritiesResource* and *UsersResource* which will be same as generated in models with Lombok and JSON annotations. Similar to other repositories such as Equipment repository, *UsersRepository* and *AuthoritiesRepository* are also generated by the generator. These two repositories are used for performing CRUD operations for users and roles (authorities). These two repositories has two additional queries for the Read/Find operations. Similar to other controllers generation described in section 4.2.2, there are some controllers need to be generated for security. Some of the security controllers are specific of resource specification and some security controllers are not specific of specification. The security controllers are described in detail below.

The main purpose of *AuthenticationController* is to provide the initial authentication to the application. This controller is annotated with RestController and RequestMapping with path as “/rest/authentication”. From the resource specification,

```
role Customer
role Works_Engineer
```

is generated as:

```
@RequestMapping("/rest/authentication")
@RestController
public class AuthenticationController {
    ObjectMapper mapper = new ObjectMapper();

    @RequestMapping(method=RequestMethod.POST)
    @Secured({"ROLE_ADMIN", "ROLE_CUSTOMER", "ROLE_WORKS_ENGINEER"})
    public String authenticate() throws JsonProcessingException {
        Object principal = SecurityContextHolder.getContext().getAuthentication().getPrincipal();
        List<String> roles = new LinkedList<String>();

        if (principal instanceof UserDetails) {
```

```

        UserDetails details = (UserDetails) principal;
        for (GrantedAuthority authority: details.getAuthorities())
            roles.add(authority.getAuthority());
    }

    Map<String, List<String>> map = new HashMap<String,
List<String>>();
    map.put("roles", roles);

    return mapper.writeValueAsString(map);
}

@ExceptionHandler(value={SecurityException.class})
@ResponseStatus(HttpStatus.UNAUTHORIZED)
public void handleSecurityException() {
}
}

```

In the above generated code, Secured annotation's parameter array is generated from "role" declaration of the resource specification. If the resource specification does not have any "role" declaration, then there will be only "ROLE_ADMIN" in parameter array which is fault user of the application.

SecurityController is used for the redirecting for the login and home page. This controller is not specific of resource specification. This class has only two methods namely login and home with parameters of path "/login" and "/"#/" respectively. When these methods are called, they redirects to the login page of the application.

```

@Controller
public class SecurityController {
    @RequestMapping("/login")
    public String login() {
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        String name = authentication.getName();
        return "static/views/login";
    }

    @RequestMapping("/#/"")
    public String home() {
        return "static/views/login";
    }
}

```

UsersController and **AuthoritiesController** are the main controllers for user management and roles management respectively. These controllers are not specific resource specification. The main purpose of UsersController is to create a user, delete the user and change the password of a user. AuthoritiesController class has only one REST method which is used to create new authority/role for the created/existing user. Similar to the generated assemblers mentioned in section 4.2.3, there are other two assemblers which are generated mandatorily in the assemblers package. These assembler Java classes are **UsersAssembler** and **AuthoritiesAssembler**. These assembler classes are not specific of resource specification. These Java classes are used to create the users and authorities.

The security configuration is needed to implement RBAC in the target application. The security configuration is implemented in *SecurityConfiguration* class which is partially specific of resource specification. This java class has four inner static classes:

- *UnauthorizedEntryPoint* - handles an unauthorized access to the application.
- *AuthenticationConfiguration* - to get the list of users and authorities from the database using JDBCAuthentication.
- *FormLoginWebSecurityConfigurationAdapter* - decides the homepage of the application and also URL authentications.
- *ApiWebSecurityConfigurationAdapter* - matches and authorizes the URL of the rest calls with '/rest'.

A preloaded data is needed for the users and authorities. This preloaded data is generated from RBAC security model. Data package is named as “.data” which is prefixed to the package name and contains one file *schema.sql*. This file contains database queries to create two tables namely users and authorities. The users table has the username and password for the application. The authorities table has the username and the role of the user. The resource specification for preloaded data is,

```
role Customer
```

```
role Works_Engineer
```

The target application will have a default user “admin”. The admin user will have access to all the URLs of the application. The “role” name(s) in the resource specification generates the queries in this file to create the user(s) with username and password. The database queries which is generated by code generator from resource specification is shown below:

```
create table if not exists users (username varchar(50) not null primary
key, password varchar(50) not null, enabled boolean not null)
```

```
create table if not exists authorities (id bigint not null primary key,
username varchar(50) not null, authority varchar(50) not null)
```

```
insert into users (username, password, enabled) SELECT 'admin', 'admin',
true where not exists (select username, password, enabled from users
where username = 'admin' and password = 'admin')
```

```
insert into authorities (id, username, authority) SELECT 1000, 'admin',
'ROLE_ADMIN' where not exists (select username, authority from authori-
ties where username = 'admin' and authority = 'ROLE_ADMIN')
```

```
insert into users (username, password, enabled) SELECT 'customer', 'cus-
tomer', true where not exists (select username, password, enabled from
users where username = 'customer' and password = 'customer')
```

```
insert into authorities (id, username, authority) SELECT 1001, 'custom-
er', 'ROLE_CUSTOMER' where not exists (select id, username, authority
from authorities where username = 'customer' and authority = 'ROLE_CUS-
TOMER')
```

```
insert into users (username, password, enabled) SELECT 'works_engineer',
'works_engineer', true where not exists (select username, password, ena-
bled from users where username = 'works_engineer' and password =
'works_engineer')
```

```
insert into authorities (id, username, authority) SELECT 1002,
'works_engineer', 'ROLE_WORKS_ENGINEER' where not exists (select id,
username, authority from authorities where username = 'works_engineer'
and authority = 'ROLE_WORKS_ENGINEER')
```

The above generated *schema.sql* should be moved to *src/main/resources* folder of the target project. The database configuration is needed to connect the database with the target application. Configurations are the setup files that are essentials for the project. Configuration package is named as “.configs” with prefix package name. This package contains *SimpleDbConfig* file which plays a main role in database configuration of the target project. The generated database configuration code uses **PostgreSQL**. So before running the target project, it is required to install PostgreSQL in the host computer.

This configuration file is not specific of resource specification. This database configuration uses JDBC authentication. The generated code has prefilled username, password, and URL as shown in Figure 4.6. These prefilled data can be replaced with other credentials.

```
String username = "postgres";
String password = "postgres";
String url = "jdbc:postgresql://localhost:5432/postgres";
```

Figure 4.6 Database credentials in SimpleDbConfig

All other parts of this file should be left unchanged. Now, the database for the target application has been configured successfully.

4.2.5 Views

The main part of MVC model is views. The views are done in HTML and AngularJS. The views are generated from ‘view’ of resource specification. If ‘view’ is not present, then the HTML and JS will not be generated. The four operations such as create, read, update and delete are generated as HTML files and functions in the JS files. The two packages such as *htmls* and *js* are generated for HTML files and JS files respectively. The CRUD operations string mentioned in the resource specification as,

```
resource Equipment on "/rest/equipments" view "CRUD"
```

The respective HTML files are generated by the software generator. The HTML views are powered by Bootstrap and AngularJS. All the generated HTML files are described below:

index file has imports of all the css and js files used in the target application. The target application uses bootstrap 3.3.6 and AngularJS 1.5.5. *login* file is the homepage of the application. It contains form fields for user authentication. On the successful authentication, it directs to *main.html*. It also has a button to sign up. *main* file is generated from resource specification. The *view* string in the resource specification creates the menus in the navigation bar. The Equipment resource view is,

```
resource Equipment on "/rest/equipments" view "crD"
```

creates the menus in the navigation bar as shown below:

```
<ul class="dropdown-menu">
  <li><a href="#/equipments/create">Create Equipment</a></li>
  <li><a href="#/equipments/create">Create Equipment</a></li>
</ul>
```

The similar code generation for purchase order as well. The menus in the navigation bar changes according to the user who has login currently. *signup* file is specific of resource specification. It contains the form fields for new user registration. The dropdown field is generated from the resource specification. The resource specification has ‘role’ declared as like,

```
role Customer
role Works_Engineer
```

then those roles are listed as options in the dropdown field as shown below:

```
<select class="form-control input-lg" required ng-init="urole='ur'" ng-
model="urole">
  <option value="ur" disabled>Select User Role</option>
  <option value="ROLE_CUSTOMER">Customer</option>
  <option value="ROLE_WORKS_ENGINEER">Works_Engineer</option>
</select>
```

If resource specification does not have ‘role’ declared, then ‘Admin’ is only option added to the dropdown. On successful account creation, it directs *login.html.restricted* file is not specific of resource specification. This file is displayed if the link is not authorized. *changePassword* file is not specific of resource specification. It contains a navigation bar and form fields to change the password for an authenticated user. On successful change of password, it redirects to login page again. *deleteAccount* file is also not specific of resource specification. It contains a navigation bar and a dialog panel to delete account message. Delete button on successful deleted account redirects to the login page. It also has the alert message box which responds with messages.

list file is specific of resource specification. If the resource specification has ‘view’ string with ‘R’ or ‘r’, then list file will be generated with the name ‘list’ and the postfix of the resource name. This file has a navigation bar and a table to list all the resource data. The table header is specific of ‘data’ block in resource specification and rows filled with JSON data from the back-end. In the purchase order resource specification,

```
data
  equipment : many Equipment rendered
  startDate : Date
  endDate : Date
  total : Long
end
```


is generated as,

```
<tr>
  <th>Equipment</th>
  <th>StartDate</th>
  <th>EndDate</th>
  <th>Total</th>
  <th>Actions</th>
</tr>
```

Similarly, the code is generated for Equipment resource as well. There is an actions column described that has accept and reject buttons. *create* file is specific of resource specification. If the resource specification has 'view' string with 'C' or 'c', then this file will be created with the name 'create' and the postfix of resource name. This file has a navigation bar and a form to create a resource. The form is generated from 'data' block of the resource specification. If a data property has 'rendered' and 'many', then a button with 'List' and postfix of respective resource name. The table row also has 'delete' button to delete the selected resource. For purchase order data specification,

```
data
  equipment : many Equipment rendered
  startDate : Date
  endDate : Date
  total : Long
end
```

From the above resource specification, the form will be generated as follows ,

```
<form class="form-horizontal">
  <div class="form-group">
    <label for="startdate" class="col-sm-2 control-label">StartDate</label>
    <input type="date" ng-model="startdate" required></input>
  </div>
  <div class="form-group">
    <label for="enddate" class="col-sm-2 control-label">EndDate</label>
    <input type="date" ng-model="enddate" required></input>
  </div>
  <div class="form-group">
    <label for="total" class="col-sm-2 control-label">Total</label>
    <input
      type="text" ng-model="total" required></input>
  </div>
  <div class="form-group">
    <label class="col-sm-2 control-label"></label>
    <button type="submit" class="btn btn-info"
      ng-click="createPurchaseOrder()">
      <strong>Create PurchaseOrder</strong>
    </button>
  </div>
</form>
```

'List Equipment' button is also generated along with the above form, because the specification has many and rendered in equipment data block. On successful creation of purchase order resource, the application will be redirected to respective list page if the user has access. If there is any field left unfilled in the form, then the resource will not be created.

update file is generated from of resource specification. If the resource specification has '**view**' string with 'U' or 'u', then this file will be created with the name 'update' and the postfix of resource name. This generated update file is same as the respective create file with a difference that this page's fields are pre-filled. On successful update of resource, the application will be redirected to respective list page. If there is any field left unfilled, then the resource item will not be updated. After all these HTML files has been generated, **index.html** is moved to *src/main/resources -> static* and all other HTML files are moved to *src/main/resources > static > views* of the target project.

JS files are generated as package '.js' with prefix of package name. Some functions in JS files are specific of resource specification. Along with the generated HTML files, the respective JS files are created. The two JS files such as **main** and **controllers** are generated in this package. **main** file is partially specific on the resource specification. If '**view**' string has any of these characters 'CRUcru' then the respective route will be created with the respective resource. The routes such as '/main', '/changePassword', '/deleteAccount', '/login' and '/signup' are created without resource specification. RoleBasedAccessService function is the main function that does the role checks. It checks by matching the roles from the login credentials with the root scope user variables. If there is an error in authorization, then it directs the path '/restricted' which has *restricted.html* view.

controllers file is partially specific of resource specification. This AngularJS file has three controllers generated which are not specific of resource specification. They are:

- 'LoginController' is used for authentication during the user login. It makes the POST rest calls to a method in **SecurityController.java**.
- 'SignupController' is used to create new users during the signup. It makes the POST rest calls to the methods in **UsersController.java** and **AuthoritiesController.java** and does the necessary operations to create new users.
- 'AccountController' is used to change password and delete user account. It makes the PUT and POST rest calls to the methods in **UsersController.java** to change password and delete account respectively.

All the above three controllers has some additional operations to validate the requests and responses. If '**view**' string in resource specification has any of characters 'CRUcru' then the respective functions such as Create, List and Update with postfixes of respective resource names will be generated inside the respective controller. After all these JS files has been generated, these JS files are moved to *src/main/resources > static > js* of the target project.

4.3 Discussion

In this section, the DSL specification and code generation has been discussed. The Equipment rental system is the scenario that had taken for the DSL specification and code generation. The scenario has been discussed with data, resource and state models at the beginning of this section. The lifecycle of purchase order such as create, accept and reject purchase orders is taken for consideration. The DSL specification for Equipment Rental System had discussed in detail in section 4.1. The DSL specification is written from data, resource, state and RBAC models which was given as an input to the software generator tool.

The software generator tool generated the code for the target application from the resource specification in section 4.2. The generated code from the resource specification has been explained in detail in respective sub sections. The generated models, resources and repositories are explained in section 4.2.1. The controllers classes generated are discussed in sub section 4.2.2. Similarly, the state model resource specification and generated assembler classes are explained briefly in section 4.2.3. The RBAC security classes are the main scope of this thesis and its code generation is explained clearly in section 4.2.4. The front-end views are important for the target application. The context about the front-end resource specification and its respective code generation was explained clearly in the last subsection. Thus, the DSL specification and its respective code generation was explained clearly in this section which implemented the target application.

The repository for the software generator tool can be found in: <https://vinodrockson@bitbucket.org/lgbanelos/secrest.git>

The repository for the generated codes of the above mentioned scenario can be found in: https://vinodrockson@bitbucket.org/vinodrockson/thesis_rentit.git

The project setup can be found in Appendix – III.

5 Case study

This chapter describes an example application that has been built using a prototypical implementation of the DSL introduced in section 4.2 and the summary. The example application demonstrates the generation of the skeleton of the back-end exposing the RESTful API, and integrating the RBAC feature in it. The purchase order-related methods in *RentIt- Equipment Rental System*, namely creating, accepting and rejecting of purchase orders, are demonstrated.

5.1 Example application (RentIt)

The full resource specification of this model is shown in Appendix-II. When the generated application is run, the login page is shown first. From our resource specification we have,

```
role Customer
```

```
role Works_Engineer
```

Then the application should be able to login with customer, works_engineer and admin. The login page of the target application is shown in the Figure 5.1. There are other options such

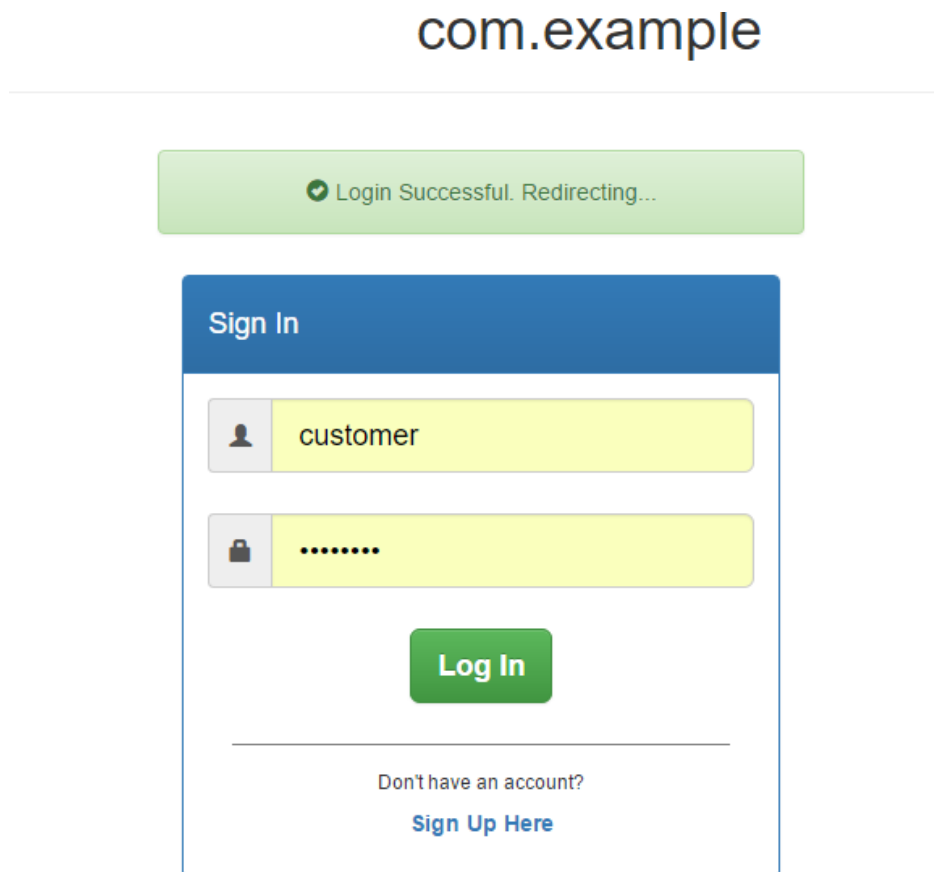


Figure 5.1 Login page of the application

as sign up, delete account and change password views and services are also generated for this application. All the views of these features are shown in Appendix-V with default user 'admin'. This shows that the application will work even without any roles.

In order to create a purchase order, we need to create an equipment. From resource specification the Equipment resource,

```
actions
  {"id" -> Equipment}

  create(Equipment) : Equipment
    with POST on "" roles [Works_Engineer]

  update(Equipment) : Equipment
    with PUT on("/{id}" roles [Works_Engineer]

  Delete() : Equipment
    with DELETE on("/{id}" roles [Works_Engineer]

end
```

Firstly, let us take create equipment and the code generated for create equipment is,

```
@Secured({ "ROLE_ADMIN", "ROLE_WORKS_ENGINEER" })
@RequestMapping(method = RequestMethod.POST, value = "")
@ResponseStatus(HttpStatus.CREATED)
public ResponseEntity<EquipmentResource> create(@RequestBody EquipmentResource equipmentResource)
    throws Exception, EquipmentNotFoundException {
    if (equipmentResource == null) {
        return null;
    }
    Equipment equipment = new Equipment();

    equipment.setName(equipmentResource.getName());
    equipment.setDescription(equipmentResource.getDescription());
    equipment.setPrice(equipmentResource.getPrice());

    Equipment createdEquipment = equipmentRepo.saveAndFlush(equipment);
    EquipmentResource res = equipmentAssembler.toResource(createdEquipment);
    HttpHeaders headers = new HttpHeaders();
    headers.setLocation(new URI(res.getId().getHref()));
    return new ResponseEntity<>(res, headers, HttpStatus.CREATED);
}
```

This shows that create () accessible only to Works_Engineer and Admin. Currently, we had login with Customer. So, if we select 'Create Equipment' from Equipment menu, it shows the restricted access as shown in Figure 5.2 because 'Customer' user cannot access this page. So, let's login again with 'Works_Engineer' user and create equipment.

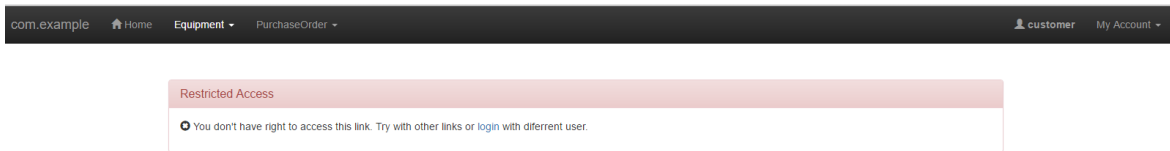


Figure 5.2 Create Equipment- Restricted access.

Since works engineer has access to it, the application allows the access to works_engineer as shown in Figure 5.3. This shows that the generated application is completely secured with role based access. The views are depend on the resource specification. The process and view is similar for update and delete operations.

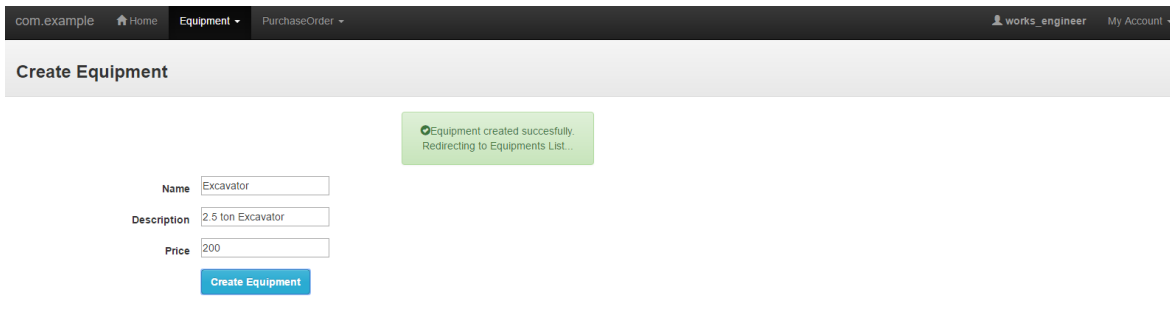


Figure 5.3 Create Equipment- Success view.

Next step is to create purchase order. From the resource specification of purchase order,

```

actions
  {"poid" -> PurchaseOrder}

  create(PurchaseOrder) : PurchaseOrder
    with POST on "/pos" roles [Customer]

  update(PurchaseOrder) : PurchaseOrder
    with PUT on "/pos/{poid}" roles [Customer]

  delete() : PurchaseOrder
    with DELETE on "/pos/{poid}" roles [Customer, Works_Engi-
neer]

  acceptPO(PurchaseOrder) : PurchaseOrder
    with POST on "pos/{poid}/accept" roles [Customer, Works_En-
gineer]
  rejectPO() : PurchaseOrder
    with DELETE on "pos/{poid}/accept" roles [Works_Engineer]

end

```

The generated code for create purchase order method is,

```

@Secured({ "ROLE_ADMIN", "ROLE_CUSTOMER" })
@RequestMapping(method = RequestMethod.POST, value = "/pos")

```

```

@ResponseStatus(HttpStatus.CREATED)
public ResponseEntity<PurchaseOrderResource> create(@RequestBody PurchaseOrderResource purchaseorderResource)
    throws Exception, PurchaseOrderNotFoundException, EquipmentNotFoundException {
    if (purchaseorderResource == null) {
        return null;
    }
    PurchaseOrder purchaseorder = new PurchaseOrder();

    List<Equipment> tempEquipmentList = new ArrayList<>();
    for (EquipmentResource equipmentResource : purchaseorderResource.getEquipment()) {
        Equipment equipment = new Equipment();
        equipment.setName(equipmentResource.getName());
        equipment.setDescription(equipmentResource.getDescription());
        equipment.setPrice(equipmentResource.getPrice());
        tempEquipmentList.add(equipment);
    }
    purchaseorder.setEquipment(tempEquipmentList);
    purchaseorder.setStartDate(purchaseorderResource.getStartDate());
    purchaseorder.setEndDate(purchaseorderResource.getEndDate());
    purchaseorder.setTotal(purchaseorderResource.getTotal());
    purchaseorder.setPurchaseorderState(PurchaseOrderState.PENDING_CONFIRMATION);

    PurchaseOrder createdPurchaseOrder = purchaseorderRepo.saveAndFlush(purchaseorder);
    PurchaseOrderResource res = purchaseorderAssembler.toResource(createdPurchaseOrder);
    HttpHeaders headers = new HttpHeaders();
    headers.setLocation(new URI(res.getId().getHref()));
    return new ResponseEntity<>(res, headers, HttpStatus.CREATED);
}

```

This shows that create method is accessible only to customer and admin. If we try to access create purchase order with present works_engineer user, we get restricted access message as shown in Figure 5.4. So, let's login again with customer user and create purchase order.

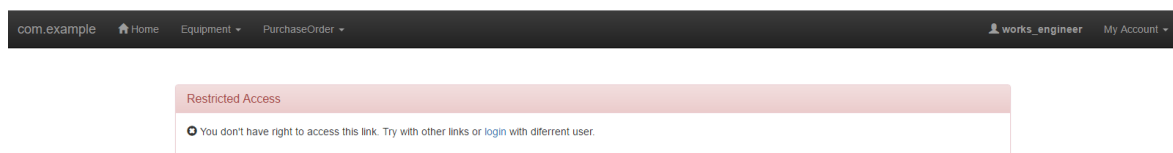


Figure 5.4 Create Purchase Order - Restricted access.

When customer user tries to create purchase order, it has been created successfully as shown in Figure 5.5. So, this evaluation about create purchase order is correct. The next step is accepting and rejecting purchase order. The generated code for accept purchase order method is,

```

@Secured({ "ROLE_ADMIN", "ROLE_CUSTOMER", "ROLE_WORKS_ENGINEER" })
@RequestMapping(method = RequestMethod.POST, value = "pos/{poid}/accept")

```

```

public ResponseEntity<PurchaseOrderResource> acceptPO(PurchaseOrder purchaseorder,
    @PathVariable("poid") Long poid) {
    PurchaseOrder temp = new PurchaseOrder();
    temp.setPurchaseorderState(PurchaseOrderState.OPEN);
    PurchaseOrderResource purchaseorderResource = purchaseorderAssembler.toResource(temp);
    return new ResponseEntity<>(purchaseorderResource,
    HttpStatus.OK);}

```

The generated code for reject purchase order method is,

```

@Secured({ "ROLE_ADMIN", "ROLE_WORKS_ENGINEER" })
@RequestMapping(method = RequestMethod.DELETE, value = "pos/{poid}/accept")
public ResponseEntity<PurchaseOrderResource> rejectPO(@PathVariable("poid") Long poid) {
    PurchaseOrder temp = new PurchaseOrder();
    temp.setPurchaseorderState(PurchaseOrderState.REJECTED);
    PurchaseOrderResource purchaseorderResource = purchaseorderAssembler.toResource(temp);
    return new ResponseEntity<>(purchaseorderResource, HttpStatus.OK);
}

```

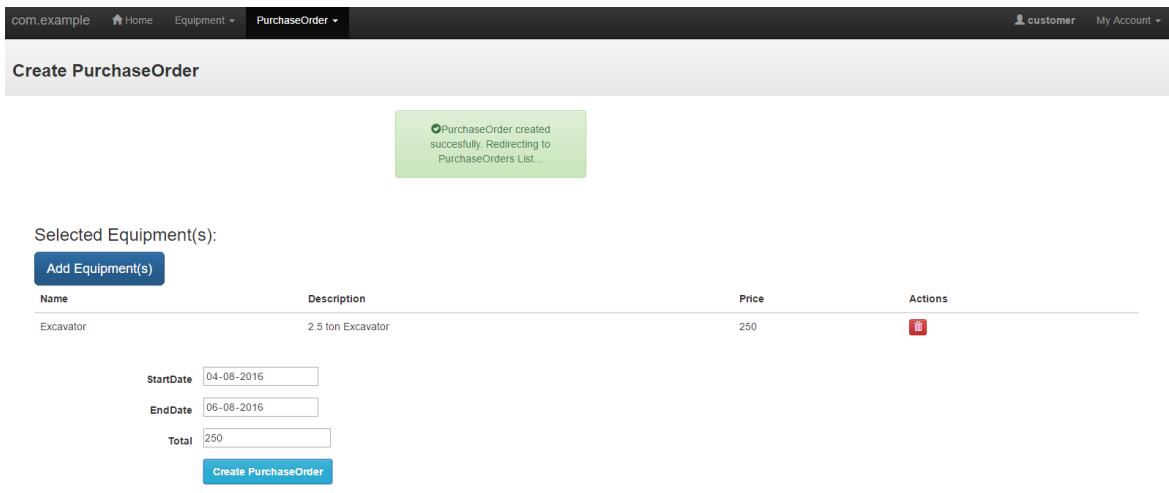


Figure 5.5 Create Purchase Order – Success view.

The generated code for acceptPO() and rejectPO() methods shows that, admin and works_engineer users have access to both these methods whereas customer user have access to only acceptPO() method. The acceptPO and rejectPO are shown as buttons in the list of purchase orders. With the current login user-customer, if we see the list of purchase order as shown in Figure 5.6, only acceptPO button is shown.

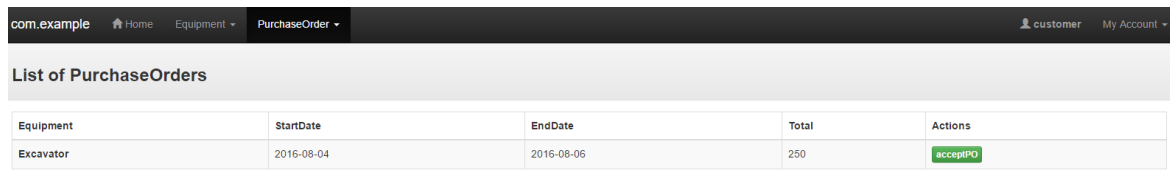
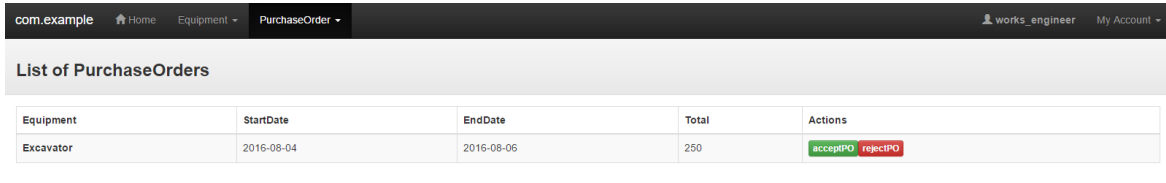


Figure 5.6 Purchase Order List – Customer user view.

If we login again with works_engineer user and see the purchase order list, both acceptPO and rejectPO buttons are shown as in Figure 5.7. This proves that the generate code and implementation for purchase order is correct as per the resource specification that is mentioned above.

Only the skeletons of the acceptPO and rejectPO methods are generated. In order to make the buttons work, the definitions of these methods has to be re-written.



Equipment	StartDate	EndDate	Total	Actions
Excavator	2016-08-04	2016-08-06	250	acceptPO rejectPO

Figure 5.7 Purchase Order List – Works_Engineer user view.

Hence, the implementation and resulted views of Equipment and Purchase Order is as per the resource specification, the validity of our approach of developed software tool is considered as successful.

5.2 Summary

The implementation of this thesis focused on the software generator which generates the REST API and the views for the application. The views are generated only for AngularJS and HTML. The implementation doesn't focus on the thyme leaf pages. This means that there are no views implemented for the thyme leaf pages which are not the view of AngularJS. The authentication and authentication through RBAC are very well working during the run time of the application. RBAC authorization generates and shows the links according to the role of the user.

6 Conclusion and future work

This chapter explains about the conclusion and the future work of this thesis. The conclusion demonstrates the work that has been done for this thesis and future shows an idea about the continuation of thesis in future.

6.1 Conclusion

The main goal of this thesis is to provide the simple way to develop an application that exposes its functionality via a REST API. To this end, a Domain Specific Language has been designed that allows developers to specify the REST API. Based on a REST API specification, it is possible to generate the skeleton of the application, including the data access layer (domain model), the controller/service layer (controller), the view layer (AngularJS user interface) and role-based access code. As a proof of concept, the code generator produces a fully functional application using Spring Boot and views in HTML which is powered by AngularJS and Bootstrap. All the necessary files that are needed for the front-end are already included in the generated view files.

The support to RBAC is added to the application by leveraging the Spring security framework. Since the front-end is powered by AngularJS, RBAC functions are also added to the AngularJS codes. The RBAC functions in JS communicate with the Spring security and provides the authentication and authorization to the API. The communications are made as REST calls. There are also front-end views and back-end code to create the users and delete existing user in the target application dynamically. All the codes that are generated are the skeleton of the RESTful API. The developer can able to add/modify the existing code. This approach will surely make the work easier for the developers by reducing the time to develop the application. It may also reduce the manpower to develop the application.

6.2 Future work

In the code generated by the code generator, there is no option of ‘forgot password’ in the login page. ‘Add/Change role’ in My Account menu doesn’t work in the current application. The future work is to optimize the existing software generator framework and add more features to the generator. The features includes:

- Generating email client with views to reset the password from the login page.
- Generating codes for message notifications with views.
- The implementation automatic code generation for role administration in RBAC [17] [18] where the super admin will have full rights on the roles of the users in the target application dynamically. The users may send the add/change roles request to the admin, the admin has to decide to accept/reject those requests.

7 References

- [1] Roy Thomas Fielding, Representational State Transfer (REST), In Architectural Styles and the Design of Network-based Software Architectures (Doctoral dissertation), University of California, Irvine), Chapter 5, 201. Retrieved from: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. [Accessed on March 19, 2016].
- [2] Brian Proffitt, What APIs Are And Why They're Important, Retrieved from: <http://readwrite.com/2013/09/19/api-defined/>. [Accessed on March 19, 2016].
- [3] Petri Selonen, From Requirements to a RESTful Web Service: Engineering Content Oriented Web Services with REST, E. Wilde and C. Pautasso (eds.), REST: From Research to Practice, 2011, pp: 259–278.doi:10.1007/978-1-4419-8303-9 11.
- [4] A. Chris Bogen and Dr. David A. Dampier, “Preparing for Large-Scale Investigations with Case Domain Modelling”, in *Digital Forensic Research Workshop (DFRWS)*, 2005.
- [5] University of Tartu's course wiki (2014 fall) - Enterprise System Integration Practicals-6 page: <https://bitbucket.org/lgbanuelos/esi-2014/wiki/Practice6>. [Accessed on April 2, 2016].
- [6] David F. Ferraiolo and D. Richard Kuhn, “Role-Based Access Controls”, in *Proceedings of the 15th National Computer Security Conference*, 1992, pp. 554 - 563.
- [7] Xin Jin, Applying Model Driven Architecture approach to Model Role Based Access Control System (Master's thesis), University of Ottawa, Canada, 2006.
- [8] Ahn G-J., Hu H., “Towards Realizing a Formal RBAC Model in Real Systems”, in *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT'07)*, 2007, pp. 215-224.
- [9] Cirit C., Buzluca F., “A UML Profile for Role-Based Access Control”, in *Proceedings of the 2nd International Conference on Security of Information and Networks (SIN'09)*, 2009, pp 83-92.
- [10] Alalfi M. H., Cordy J. R., Dean T. R., “Recovering Role-Based Access Control Security Models from Dynamic Web Applications”, in *Proceedings of the 12th International Conference on Web Engineering (ICWE'12)*, 2012, pp 121-136.
- [11] Tark K., Role Based Access Model in XML based Documents (Master's thesis), University of Tartu, Estonia, 2013.
- [12] Vitaliy Schreibmann and Peter Braun, “Model-driven Development of RESTful APIs”, in *Proceedings of the 11th International Conference on Web Information Systems and Technologies*, 2015.
- [13] Tobias Fertig and Peter Braun, “Model-driven Testing of RESTful APIs”, in *WWW 2015 Companion*, 2015.
- [14] Jonathan Robie, Rob Cavicchio, Remon Sinnema and Erik Wilde, “Describing RESTful Services Without Tight Coupling”, in *Proceedings of Balisage: The Markup Conference*, 2013.

- [15] Arie van Deursen and Paul Klint, “Domain-Specific Language Design Requires Feature Descriptions”, *Journal of Computing and Information Technology-CIT*, 2002.
- [16] Marjan Mernik, Jan Heering, and Anthony M. Sloane, When and how to develop domain-specific languages, *ACM Computing Surveys*, 37(4): 316–344, 2005.doi:10.1145/1118890.1118892.
- [17] Andras Belokosztolszki, Role-based access control policy administration, University of Cambridge's Technical Report: 586, 2004.
- [18] Ravi S. Sandhu, Edward G. Coyne, Hal L. Feinstein and Charles E. Youmank, Role-Based Access Control Models, IEEE Computer Society Press Los Alamitos, Volume 29 Issue 2, February 1996.
- [19] Fielding, R. T.; Taylor, R. N. (2000), Principled design of the Modern Web Architecture: 407–416. doi:10.1145/337180.337228.
- [20] Learn REST: A Tutorial: <http://rest.elkstein.org/>. [Accessed on April 7, 2016].
- [21] Plant Hire System’s Scenario: https://courses.cs.ut.ee/MTAT.03.229/2014_fall/uploads/Main/PlantHireScenario.pdf. [Accessed on March 20, 2016].

Appendix

I. Grammar implementation

The first step of the implementation is writing the DSL grammar for the code generation. From this grammar, we can able to write the Xtend code for software generator. The grammer should be written in Xtext file. Grammar implementation is shown below:

Grammar:

```
grammar org.xtext.example.rsec.RSec with org.eclipse.xtext.common.Terminals
```

```
generate rSec "http://www.xtext.org/example/rsec/RSec"
```

```
ResourceSpecification :  
    {ResourceSpecification}  
    'package' packageName=QualifiedName  
    (imports+=Import+)?  
    (elements+=Type)*;
```

```
Type:  
    DataType | ResourceType | Roles;
```

```
DataType:  
    'datatype' name=ID;
```

```
Import:  
    'import' importedNamespace=QualifiedNameWithWildcard;
```

```
QualifiedNameWithWildcard:  
    QualifiedName '.*'?
```

```
QualifiedName:  
    ID ('.' ID)*;
```

```
Roles:  
    'role' name=ID;
```

```
ResourceType :  
    'resource' name=ID ('on' path=STRING)? ('view' views=STRING)?  
    ('data'  
        properties+=Property+  
    'end')?  
    ('actions'  
        ('{' pathvariable += STRING '->' resource += [Re-  
sourceType]  
        (',' pathvariable += STRING '->' resource +=  
[ResourceType])*}')?  
        events+=Event+  
    'end')?  
    ('states'  
        states+=State*  
    'end')?  
    'end'  
;
```

```
Event:
```

```

    name=ID '(' (paramType = [ResourceType])? ')' ':' (many ?= 'many')?
    returnType=[Type] 'with' verb=ID 'on' path=STRING (roles ?=
'roles')?
    ('['mrole += [Roles](',' mrole += [Roles])*']')?
;

Property:
    name=ID ':' (many ?= 'many')? type=[Type] (render ?= 'rendered')?
;

State:
    'state' name=ID
        transitions+=Transition*
    'end'
;

Transition:
    event=[Event] '=>' state=[State]
;

```

The description for the above grammar is as follows:

1. **grammar** specifies what type of grammar that we want to use in the project `org.xtext.example.rsec.RSec`. The type of grammar that we specify here will decide the type of generator in Xtend and other files in the project.
2. `ResourceSpecification` is like a container that consist of all the elements that needed for resource specification. The rule comprises of package name of the target project, imports of the packages and the `Type`.
3. `Type` holds the `Datatype`, `ResourceType` and `Roles`.
4. `Datatype` has the prefix “`datatype`” followed by name of the data type. The `datatype` is used in data to specify the type of the data.
5. Similar to `Datatype`, `Import` has the prefix followed by the package name of the `Datatype`.
6. `Roles` is for the security purpose. It is used of declaring the roles used in the target application. It has a prefix “`role`” followed by role name.
7. `ResourceType` is the main part of this DSL. It consists of data, actions and states. Each of these has separate rules.
8. `Event` is the set of rule for actions. The actions are methods where the specific operations can be performed. Every event has a name with input parameter type “`paramType`” and the return type “`returnType`”. If there are many return types, it can be prefixed by “`many`”. The REST type is “`verb`” and path of the action is defined in “`path`”. Roles for which the action can be accessed is defined by “`roles`” and `mrole`.
9. `Property` is the set of rules for defining the data. The type of the data type can be “`many`”. The “`rendered`” is used when the specific data is used in the view.
10. `State` is the set of rules for states. Each state has many transitions and each state has a name which has a prefix “`state`” and closed by “`end`”.
11. Each `Transition` consists of event name implies to a state name.

II. DSL specification

```
package com.example

import java.lang.Boolean

datatype String
datatype void
datatype BigDecimal
datatype Date
datatype Long
datatype Boolean

role Customer
role Works_Engineer

resource Equipment on "/rest/equipments" view "crD"
  data
    name: String
    description: String
    price: BigDecimal
  end

  actions
    {"id" -> Equipment}

    create(Equipment) : Equipment
      with POST on "" roles [Works_Engineer]

    update(Equipment) : Equipment
      with PUT on "{id}" roles [Works_Engineer]

    Delete() : Equipment
      with DELETE on "{id}" roles [Works_Engineer]
  end
end

resource PurchaseOrder on "/rest/orders" view "CRUD"
  data
    equipment : many Equipment rendered
    startDate : Date
    endDate : Date
    total : Long
  end

  actions
    {"poid" -> PurchaseOrder}

    create(PurchaseOrder) : PurchaseOrder
      with POST on "/pos" roles [Customer]

    update(PurchaseOrder) : PurchaseOrder
      with PUT on "/pos/{poid}" roles [Customer]

    delete() : PurchaseOrder
      with DELETE on "/pos/{poid}" roles [Customer,
Works_Engineer]

    acceptPO(PurchaseOrder) : PurchaseOrder
      with POST on "pos/{poid}/accept" roles [Customer,
Works_Engineer]
    rejectPO() : PurchaseOrder
```

```

                                with DELETE on "pos/{poid}/accept" roles [Works_Engi-
neer]
    end
    states
        state initial
            create => pending_confirmation
        end
        state pending_confirmation
            acceptPO => open
            rejectPO => rejected
        end
        state rejected
            update => pending_confirmation
        end
        state open
            delete => closed
        end
        state closed
        end
    end
end
end

```


III. Project setup

Running the Xtext project will lead to the new Eclipse application. Since it is very complex to create a Spring Boot project directly in Eclipse, the better solution is to import a new Spring Boot project in eclipse. Before this, it is better to install Spring software plugin in Eclipse. The first step in project setup is to add the **Xtend Library** in to the project and make sure that the project has **JRE system library** and **Maven Dependencies**. The next important step is to add the dependencies in “pom.xml”. The list of mandatory dependencies that are needed are shown below. Sometimes Lombok dependency and other dependencies may not get effect from pom.xml. In this case, it is better to add the missing dependencies as an external JARs.

Maven Dependencies:

```
<dependencies>
  <dependency>
    <groupId>commons-dbcp</groupId>
    <artifactId>commons-dbcp</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.16.8</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.hateoas</groupId>
    <artifactId>spring-hateoas</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>
</dependencies>
```

Make a new folder in your target project and create a new file in that folder with extension “.rsec”. The resource specification should be written in .rsec file. If you save this .rsec file, a new folder “src-gen” will be created. This folder should be converted as a source folder.

‘*application.properties*’ is an important file which will not be automatically generated. It defines the general properties of the project. It will be compiled along with configuration files during the compilation time. This file can be found under *src/main/resources* folder. If this file is not present, then it is better to create this file under this folder. The below mentioned application properties should be added to this file. This database setup will re-start the database on every new run of the application which deletes the old data present in the tables. To change this re-start, ‘create-drop’ should be replaced with ‘update’.

application.properties:

```
spring.thymeleaf.cache = false
spring.jpa.database=POSTGRESQL
spring.jpa.show-sql=false
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.properties.hibernate.hbm2ddl.import_files=schema.sql
```

After the code generation, the sql, html and js files should be moved to the specified locations. If there is any need for code changes in the generated files, it can be done manually and saved in a newly created source folder inside a new package. If resource specification is changed and saved, then it will re-create all the files/packages in *src-gen* source folder which in turn overwrites the changes in files if made already. The files which depend on this modified file may show some import errors because its location has been changed. It is better to edit those files also in order to avoid compilation error. If it is needed to format the generated code of a file, then go to that specific file and press ‘**Ctrl + Shift + F**’.

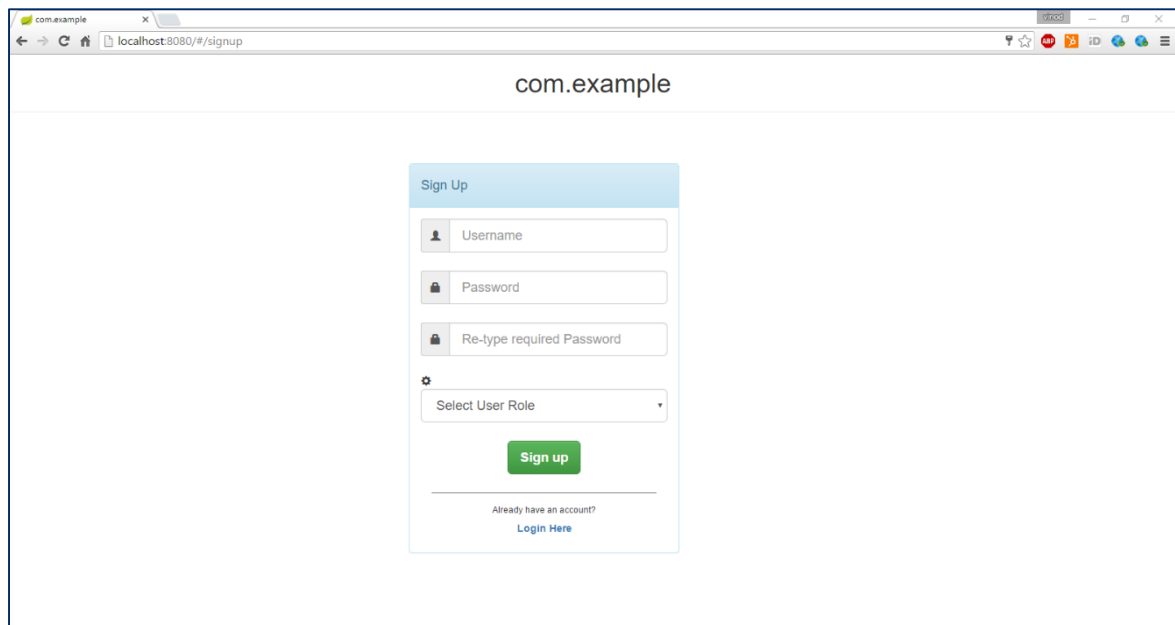
The final stage of implementation is compiling and running the application. This can be done by:

- The application can be run by *src/main/java -> packagename -> Application.java*
- Right click on this file -> Run as -> Java Application
- Now the application will start compiling and run in port 8080 (default).
- Then, go to in the web browser.

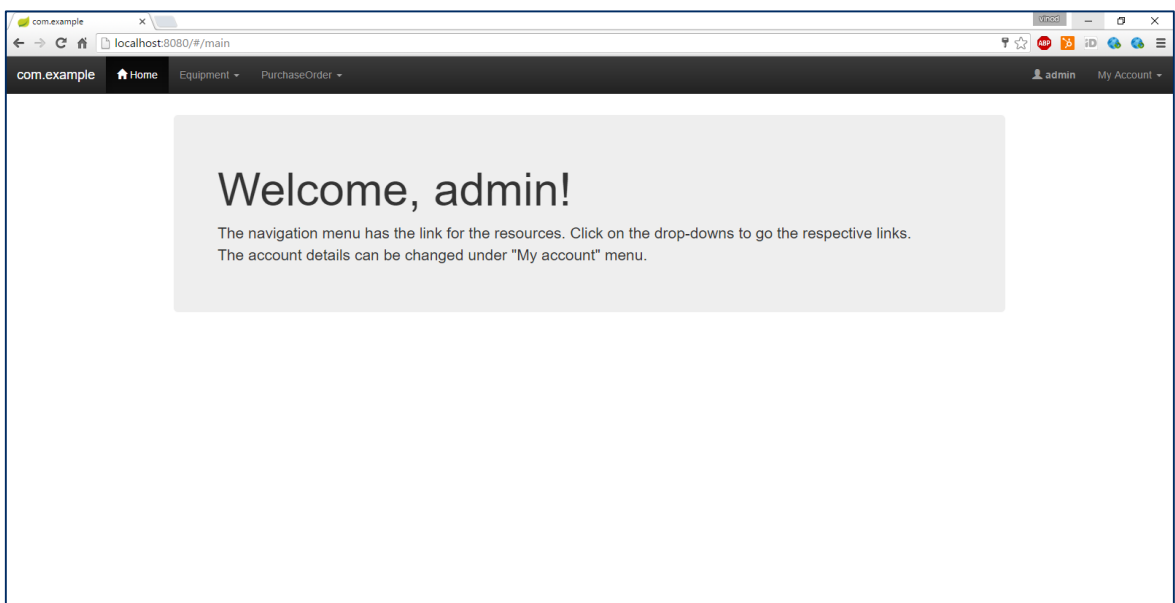
If successful application will show the login page. It can be login with any credentials given in ‘**role**’ of resource specification with lowercase letters. The username and password will be same at the beginning. If no ‘**role**’ is used in resource specification, then ‘admin’ can be used to login.

IV. Results of application – Views

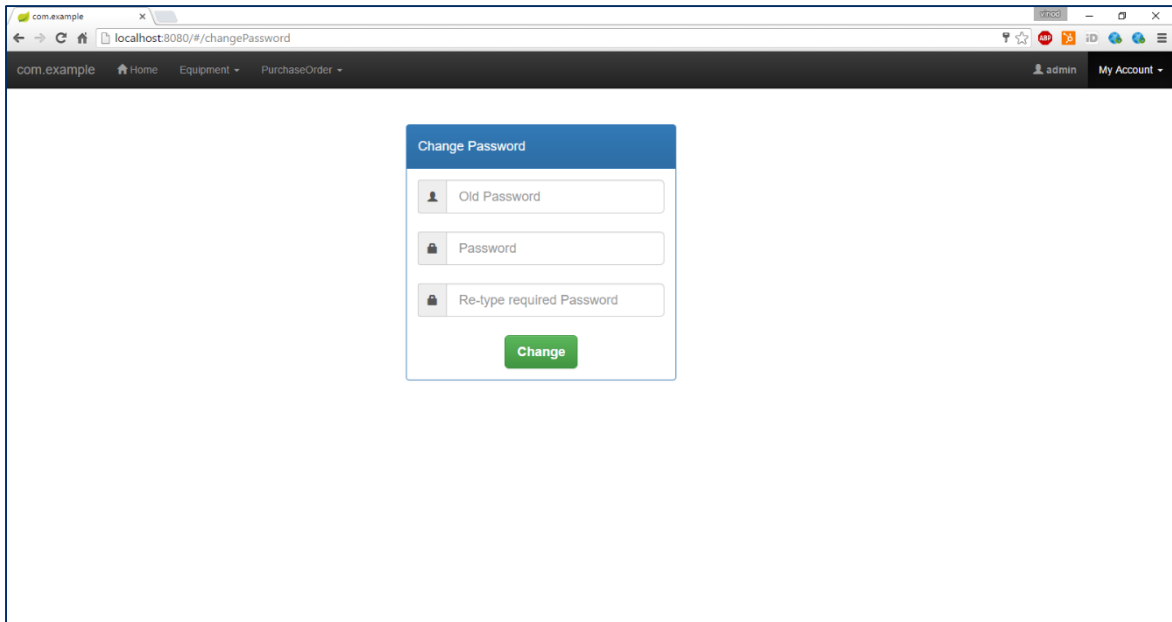
A. Generated Sign up page



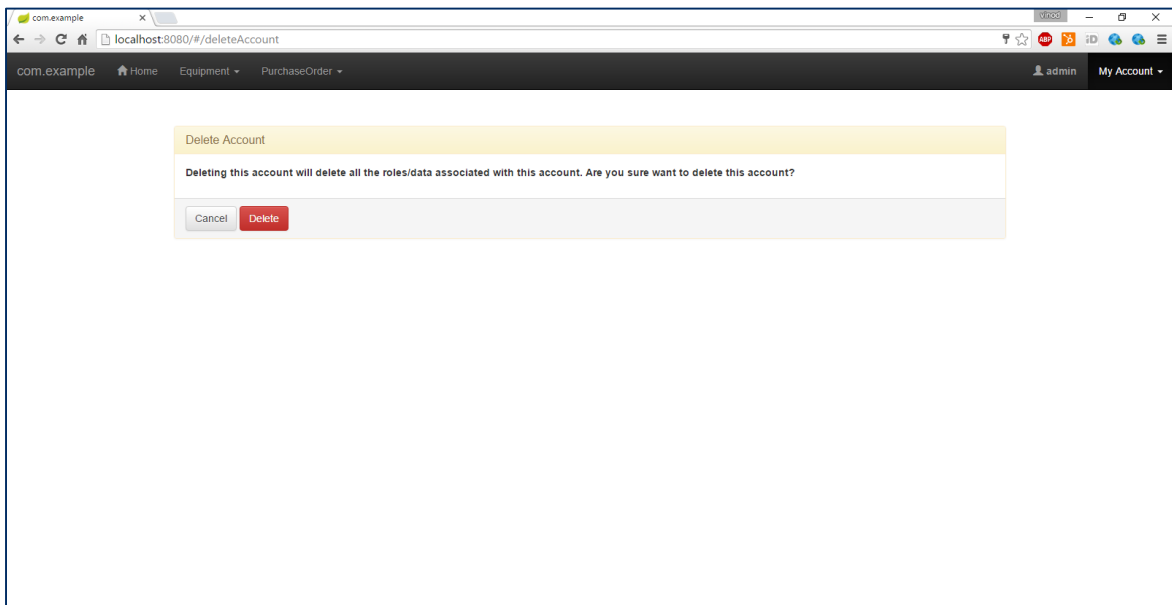
B. Generated Main page for admin user



C. Generated Change Password page



D. Generated Delete account page



V. License

Non-exclusive licence to reproduce thesis and make thesis public

I, Vinod Infant Dass John Rozario,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:
 - 1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and
 - 1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis **Model-based Role Based Access Control for RESTful Spring applications**, supervised by Dr. 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, **09.08.2016**