UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Philip John

Automated Testing of Hypermedia REST Applications

Master's Thesis (30 ECTS)

Supervisor(s): Luciano García-Bañuelos

Tartu 2016

Automated Testing of Hypermedia REST Applications

Abstract:

Testing is one essential part of the software development lifecycle and Test Driven Development is one of the main practices in agile methodology. During the development of a RESTful web application, developers oftentimes focus only in testing the business logic and neglect testing the protocol implementing REST interactions. In this context, we propose a tool to automate the generation of test cases that exercise the interactions required by a RESTful application. The tool takes as input user stories written in restricted version of Gherkin, a widely use domain specific language for behaviour driven development. User stories written in this variant of Gherkin capture the essence of the interactions required by a REST application in a way that it is possible to derive test cases from them. Moreover, the tool derives fully functional mock implementations from the same input user story. Such mock implementations can be then used by programmers to develop the client side without requiring the actual implementation of the REST application. This document introduces the design principles and implementation of the tool and presents a study case showcasing its use.

Keywords:

Test-Driven Development, RESTful Applications, Domain Specific Languages

CERCS: P170 - Computer science, numerical analysis, systems, control

Hypermedia REST rakenduste automatiseeritud testimine

Lühikokkuvõte:

Testimine on oluline osa tarkvaraarenduse elutsüklis ja testidel põhinev arendamine on üks peamistest praktikatest Agile metoodikas. Tihti keskenduvad programmeerijad RESTful rakenduse loomise protsessis äriloogika testimisele ja unustavad kontrollida protokolli, mis teostab REST interaktsioone. Selles kontekstis pakutakse välja tööriist, mis automatiseerib testide genereerimist ja teostab interaktsioone RESTful rakendusega. Tööriist võtab sisendiks kasutuslood, mis on koostatud Gherkini kitsendatud versiooniga. See on domeenispetsiifiline keel käitumispõhiseks arenduseks. Kasutuslood, mis on kirjutatud selles Gherkini variandis, hõlmavad REST rakenduse poolt nõutud interaktsioone sellisel viisil, et neist on võimalik genereerida teste. Veelgi enam, tööriist genereerib samalt kasutusloolt täisfunktsionaalse pseudoteostuse. Programmeerijad saavad kasutada neid pseudoteostusi kliendipoole arendamiseks, vajamata REST rakenduse tegelikku teostust. Käesolev töö tutvustab tööriista kasutust ja disainiprintsiipe ning esitab näite selle kasutamisest.

Võtmesõnad:

Testidel põhinev arendamine, RESTful rakendused, domeeni spetsiifiline keel

CERCS: P170 - Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

1 Introduction

1.1 Context

The number of web services is continuously increasing of which many of their distributed systems uses the architectural style called Representational State Transfer (REST). The style was suggested by Fielding in [1]. Given its simplicity and proved advantages, the software industry has widely adopted the REST architecture in the development of not only web-based applications but also other contexts, such as mobile applications.

In contrast to following the constraints and developing RESTful APIs, there is very limited research based on the quality assurance of the corresponding services. One of the methods to improve the quality of a REST service is by providing the quality assurance in the literature about the development of those services [2]. Another method is to provide automatic test case generation which can lead to lower development costs. In the development of a distributed application using RESTful architecture, the developers come up with test cases to validate the functioning of the application. Writing tests is costly and oftentimes developers focus mainly on writing tests for business logic instead of writing tests for RESTful interactions. Also, manually writing test cases is subjected to human errors. Providing automatic test case generation would avoid manual writing of test cases by the developers and thereby reduce the cost and reduce the chance of human errors.

1.2 Proposed Approach

This work tackles the problem of test case generation from scenarios using Model-Driven Testing approach. From the design point of view, we see a RESTful application as consisting of two aspects: a structural aspect, which deals with the data structure of the resources exposed by the application, as well as CRUD operations over these resources (i.e. a resource model), and a dynamic part, which deals with determining which operations can be applied to a resource given its current state (i.e. a resource lifecycle model). The former aspect is usually captured by means of annotated class diagrams while the latter can be captured by means of state chart diagrams. We contend that existing tools (e.g. Apiary blueprint-related, swagger-related, etc.) cover only the structural aspect.

The thesis proposes a tool to automate the generation of test cases that exercise the application by providing class diagrams and state charts in the form of textual Domain Specific Languages (DSLs). More concretely, we design domain specific languages embarked as a tool that produces test code for Java applications using the Spring MVC framework. We consider Gherkin language as the DSL used as it could capture the structural as well as behavioural aspects of the application. Moreover, the Gherkin language can be written in plain spoken language that can be understood by end users. Using this tool, the developers have to provide the resource and its lifecycle models in the form of a Gherkin feature file and it will generate the test cases automatically. Additionally, we provide the generation of a mock controller which are tested by default with the generated test cases. We contend that the generated test cases using our approach alleviate not only the effort in writing the test code but also in bringing the benefits of TDD, by checking that all the RESTful interactions are properly implemented. Unlike the existing tools that cover only the structural aspects of a hypermedia REST application, this solution we propose covers the application's structural as well as dynamic aspects.

1.3 Objectives

From the above, we identify the following general objective:

> *Developing a domain specific language that allows programmers to specify the interaction protocol of a RESTful application and the set of tools to generate testing artefacts from the interaction protocols thus specified.*

The above can be further refined in the following specific objectives:

- Designing a domain specific language for specifying RESTful interactions
- Implementing code generators for:
    - Test case suites for exercising the protocol implementing the RESTful interactions
    - A mock testing implementation that mimics the actual RESTful application according to the examples specified using the domain specific language

As a proof concept, we will consider code generation for RESTful applications written with Spring Boot framework. The latter implies that we target applications written in Java and more specifically, enterprise applications built on top of Spring MVC framework.

1.4 Document Outline

The rest of this thesis report is structured like the following.

Chapter 2 introduces the various theories and concepts which will be used to discuss the approach throughout the thesis report.

Chapter 3 summarizes the various researches related to the context under discussion and explains the existing tools providing similar functionalities as our aimed tool.

Chapter 4 provides an overview of the architecture used in the approach and provides an initial idea about the planned method of implementation of the tool.

Chapter 5 is probably the most important chapter of the thesis as a tool will be developed step by step using the approach discussed.

In chapter 6, the implemented tool will be evaluated in order to verify how much the tool can be used to solve real-world problems. It also provides the existing limitations of the tool.

Chapter 7 summarizes the thesis as a whole, providing the final outcomes from the implementation and validation perspective. It also provides some suggestions for future work.

2 Background

In this chapter, some of the important theory and concepts, based on which the thesis discussion will be progressed, are briefly described. First, we discuss the main concept of this thesis, which is Representational State Transfer or REST and its principles. Then we introduce the concept of Test-Driven Development. The discussion is followed by a brief introduction to class diagrams and statechart diagrams. Then we introduce the concept of Domain Specific Languages (DSL) with examples explaining how we can represent class diagrams and state diagrams in the form of simple DSLs. Finally, an introduction to Gherkin language structure is briefly explained.

2.1 Representation State Transfer (REST)

REST or Representational State Transfer is an architectural style proposed by Fielding [1] that consists of a set of constraints. A REST service is a web of interconnected resources, based on a hypermedia model that determines possible resource state transitions along with the relationships between the resources. The clients discover which controls to execute at runtime. This constraint is named as HATEOAS [Hypermedia As The Engine Of Application State]. As a result, it is possible to provide a finite set of valid URIs to the web services as their entry points [3]. The services built on these constraints have a Resource-Oriented-Architecture (ROA) [4]. In order to design and test a ReSTful API, it is important to find the resources and their relationships so that uniform operations can be selected for each resource, and can define the data formats for them.

Before knowing the principles behind REST, it is required to understand the various components used in a REST API.

Http request and HTTP response

In a RESTful system, the clients and servers interact by sending messages across each other following a predefined protocol. In the case of web APIs, this protocol is HTTP (HyperText Transfer Protocol). The client sends an HTTP request to the server and the server responds with an HTTP response.

HTTP is a request-response based protocol. The client initiates the communication by sending an HTTP request and the server will respond back with HTTP response. The structure of HTTP request and HTTP response is explained below.

HTTP request has three main components [5].

- Request Method, URI and Protocol Version – This constitutes the first line of the request. It contains the HTTP Request Method, followed by the URI to the method, and the HTTP protocol name with the version used.
- Request Header – It is used to communicate information regarding the client environment. Some of the common headers are Content-Type, Content-Length, Host etc.
- Request Body – This is the actual request which is being sent to the server. The headers and body are separated by a blank line. In our context, for REST, the request body is sent in the form of a JSON string.

Similarly, Http response also has three main components [5] which are as follows.

- Protocol Version, Status Code and Short description – The first line of a HTTP response contains the protocol name and version, the status code of the request and a short description of the status code. A status code 200 would mean that the request is successful and the description would be 'OK'. A status 404 would mean that the request was not found and the description would be 'Not Found'.
- Response Headers – These are similar to request headers, except that request header would provide the client environment information while the response header gives the server environment information. For example, Content-Type informs the client how to understand the response body.
- Response Body – This is the actual response which is rendered on the client window. Similar to the request body, in our context, the request body has a JSON structure.

Principles of REST

According to Fielding [1], REST is based on four principles which are as follows.

Principle 1: Resource identification through URI – The first and main principle of REST to think in terms of resources rather than physical files. These resources are identified using URIs which are used for the discovery of resources and their corresponding services.

Some examples of resources with URI are:

- www.example.com/image/image.jpg (Image resource)
- www.example.com/customers/10001 (Dynamically pulled resource)
- www.example.com/videos/v10001 (Video resource)
- www.example.com/home.html (Static resource)

Principle 2: Uniform interface – It says to keep the interfaces uniform and simple. This can be achieved by combining the uniform methods of HTTP protocol with the resource operation. The various HTTP methods are GET (getting a resource), POST (creates or submits the resource), PUT (updates the resource) and DELETE (deletes the resource). By combining the HTTP methods with the resource names, we can create uniform interfaces leading to simplified communication. The principle is illustrated in Table 2.1.

Ordinary method names	**HTTP methods**	**REST uniform URL**
createPurchaseOrder	POST	rest/pos
getPurchaseOrders	GET	rest/pos
getPurchaseOrder	GET	rest/pos/10001
updatePurchaseOrder	PUT	rest/pos/10001
removePurchaseOrder	DELETE	rest/pos/10001

Table 2.1 Sample method names, HTTP methods and URLs

Principle 3: Self-descriptive messages – The metadata of the resources are used and therefore, the resources are decoupled from their representation and their content can be accessed in various formats. To denote the request and the response, some kind of representation is used, which is JSON in our context. For example, below shows a simple JSON snippet for creating a new plant with name, description and price.

```
{
   "name":"Mini excavator",
   "description":"Excavator 1.5 tons",
   "price":100.0
}
```

A successful creation results in a response body like the following with HTTP status code 201 (created) and a generated id value.

```
{
   "id":1,
   "name":"Mini excavator",
   "description":"Excavator 1.5 tons",
   "price":100.0
}
```

Principle 4: Stateful interactions through hyperlinks – The interactions with the resources are stateless. The interactions are based on explicit state transfer. Every request is independent. The server need not keep track of the previous requests.

2.2 Test-Driven Development

Test-driven development is a programming practise that lets the developers to write additional code only when an automated test had failed and to eliminate duplication [6]. The actual goal of TDD is to write working clean code. The conventional application development cycle performs coding first, then testing and finally commit. Developers following the TDD approach make an effective adjustment. They perform testing first, coding second and commit. The process is repeated multiple times till the corresponding tests are passing. In other words, the application design is driven by the test.

TDD approach also aims at eliminating duplication. In other words, the written code should not only be testable but it should also be maintainable [6]. Once the test is passed, effort should be put into refactoring the code. Eliminating duplication results in increased cohesion decreased dependency, which are the core aspects of a maintainable code. The biggest advantage of using TDD approach is that a well-structured and test backed code is easier and safer to change.

We will consider an example of Purchase Order - the implementation of the creation of Purchase Order using TDD approach. Initially, we write the test case for the creation of PurchaseOrder. The JUnit test case of the functionality would look like follows.

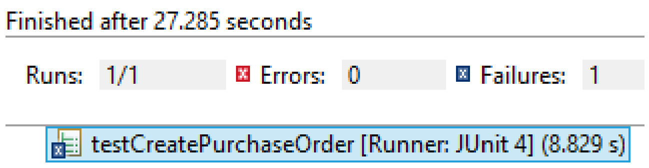
```
@Test
public void testCreatePurchaseOrder() throws Exception {
  Plant plant = new Plant();
  plant.set_id(new Long(1));

  PurchaseOrder purchaseOrder = new PurchaseOrder();
  purchaseOrder.setPlant(plant);
  purchaseOrder.setStartDate(LocalDate.of(2016, 05, 01));
  purchaseOrder.setStartDate(LocalDate.of(2016, 05, 04));

  MvcResult result = mockMvc.perform(post("/rest/pos")
          .content(mapper.writeValueAsString(purchaseOrder))
          .contentType(MediaType.APPLICATION_JSON))
          .andExpect(status().is(201))
          .andReturn();

  purchaseOrder = mapper.readValue(result.getResponse()
                      .getContentAsString(), PurchaseOrder.class);
  Assert.assertThat(purchaseOrder.getPoStatus(),
                      equalTo(POStatus.PENDING));
}
```

On executing this test case, we would get the following result in Eclipse which implies the test has failed.



```
Finished after 27.285 seconds

 Runs:  1/1     Errors:  0     Failures:  1

    testCreatePurchaseOrder [Runner: JUnit 4] (8.829 s)
```

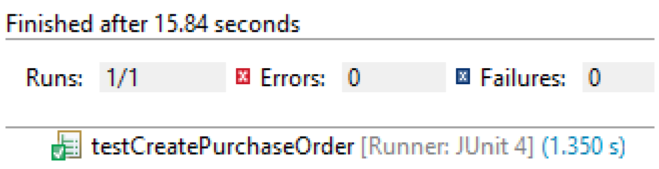
The test has failed because the method corresponding to `POST /rest/pos` is not found. As a result, we would provide the basic structure of the method.

```
@RequestMapping(value="/pos",method=RequestMethod.POST)
public ResponseEntity<PurchaseOrder> createPO(
      @RequestBody PurchaseOrder po) {
      HttpHeaders headers = new HttpHeaders();
      return new ResponseEntity<PurchaseOrder>(po, headers,
            HttpStatus.valueOf(201));
}
```

We execute the test again. Again the test is failed due to assertion error saying that it expected a status of “PENDING” but obtained was `null`. As a result, we provide the complete implementation of the creation of PurchaseOrder like follows.

```
@RequestMapping(value="/pos",method=RequestMethod.POST)
public ResponseEntity<PurchaseOrder> createPO(
      @RequestBody PurchaseOrder po) {
      po.setPoStatus(POStatus.PENDING);
      po = purchaseOrderRepo.save(po);
      HttpHeaders headers = new HttpHeaders();
      return new ResponseEntity<PurchaseOrder>(po, headers,
            HttpStatus.valueOf(201));
}
```

The tests are run again.


```
Finished after 15.84 seconds
Runs: 1/1    Errors: 0    Failures: 0
testCreatePurchaseOrder [Runner: JUnit 4] (1.350 s)
```

As seen above, we got the tests passing. To summarize, we implemented the creation of Purchase Order functionality using TDD approach by writing a failing test initially and then by implementing the functionality by repetitive development and testing process.

2.3 Class diagram and State chart diagram

A class diagram is a UML static structure diagram that describes an application using its classes, attributes and operations, and the relationships between those classes. A class in a class diagram is represented by a box with three rectangular boxes inside it. The top box provides the name of the class, the middle rectangle contains the attributes in the class and the lower box contains the operations declared within the class. The classes will be related or associated to other classes, which are depicted by lines between them.

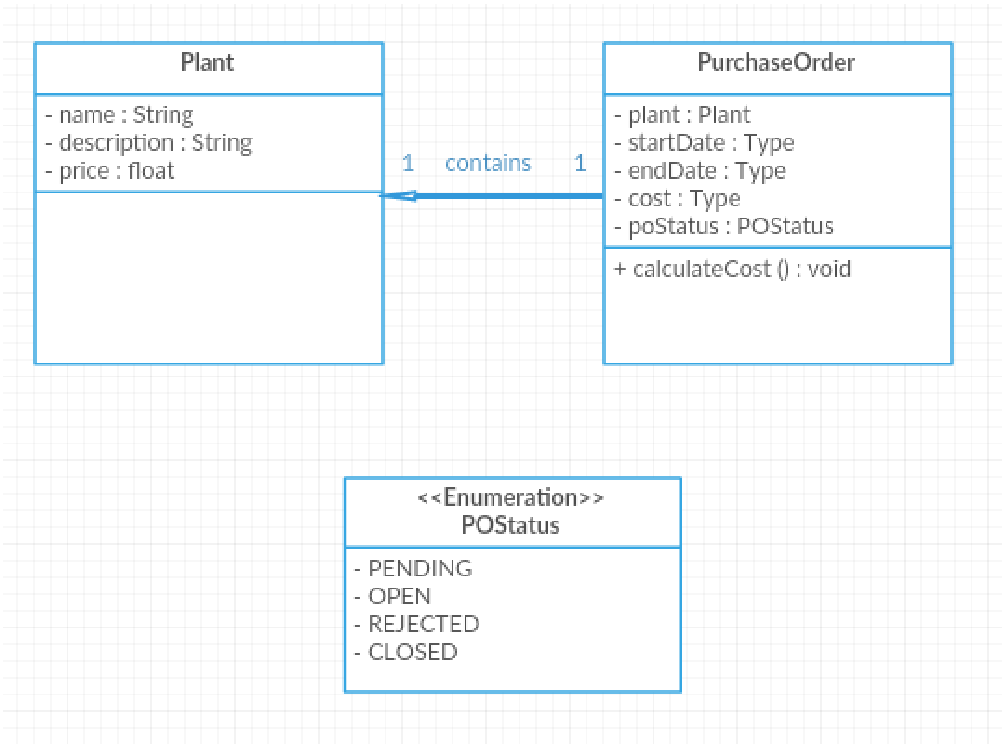


Figure 2.1: Class diagram example

Figure 2.1 above shows a basic class diagram of a Purchase Order scenario. The class diagram shows two classes (PurchaseOrder and Plant) and one enumeration (POStatus). It shows the various relationships that exist between the classes. For example, Plant has a “contains” relationship with PurchaseOrder which is a one-to-one relationship. The POStatus enumeration provides the various statuses of PurchaseOrder.

A state chart diagram is a representation of a state machine that visualises the change of states of a modelled element. It shows which activities are executed based on the occurrences of events. It displays the various states an object goes through in its life, based on an external event [7]. States, events and transitions constitute a state diagram. States are those values which certain attributes of an object possess. An object continues to be in a single state until an event is triggered on it. An event is any kind of occurrence applied to the object which may or may not change its state. The same event can have multiple effects for various states. A relationship between two states is termed as a transition which indicates a change from a state to another state [8].

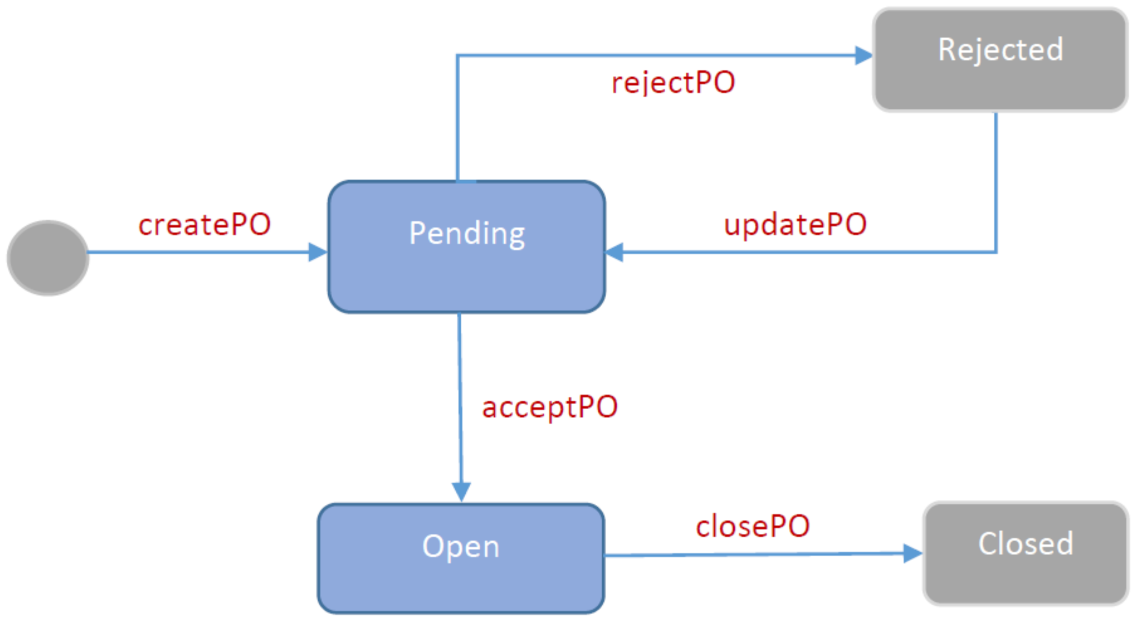


Figure 2.2: Statechart diagram

The statechart diagram for a Purchase Order is shown in Figure 2.2. The state diagram displays various transitions Purchase Order could go through from one state to another. For example, if the Purchase Order is in “Pending” state, it can either go to “Open” or “Rejected” states. When the clerk calls `acceptPO` method, the status would become “Open” and if he calls `rejectPO` method, then the status would be “Rejected”.

2.4 Domain Specific Language (DSL)

A domain specific language (DSL) is a small programming language or execution specific language which provides a notation towards the application domain and is focused on some concepts and features of that particular domain [9]. The domain can be anything. SQL and XMLs are examples of domain specific languages. A DSL can be used to generate various contents of a system in an application domain. A well-designed DSL is based on a clear understanding of the underlying application domain so that the required contents can be generated easily [9]. One example of textual DSL representation of the class diagram in Figure 2.1 is represented below.

```
package domain.model {
	enum POStatus {}

	entity Plant {
		name: String
		description: String
		price: Double
	}

	entity PurchaseOrder {
		plant: Plant
		startDate: Date
		endDate: Date
		cost: Double
		poStatus: POStatus
	}
}
```

Similarly, the state chart diagram in Figure 2.2 can be denoted in the form of DSLs. One such example is provided below.

```
events
	createPO
	acceptPO
	rejectPO
	updatePO
	closePO
end

state empty
	createPO => pending
end

state pending
	acceptPO => open
	rejectPO => rejected
end

state rejected
	updatePO => pending
end

state open
	closePO => closed
end

state closed
end
```

Since a DSL can take any structure, we can model it as a Gherkin feature and provide the characteristics of a state chart model and other details in the feature file.

2.5 Gherkin language

We consider Gherkin language as the DSL in the approach as a Gherkin feature can be used to represent the resource associations and the corresponding state transitions. Also, it is ideal to use an existing language rather than generate a completely new DSL as it can be reused for other purposes as well.

Gherkin is the language used for writing Cucumber features. The biggest advantage of using Gherkin is that it is readable not only by the computer but also the stakeholders as it is written using plain spoken language [10]. The keywords used in Gherkin has been translated into over forty languages. It is not tied down to any particular spoken language. As a result, we can say that even though it is considered as a programming language, its primary goal is human readability. It means that we can write automated tests that can be read like a documentation. A small Gherkin example is given below.

```
Feature: PurchaseOrder feature
      As a customer
      In order to rent plant equipment
      I need to process a Purchase Order

Scenario: Creation of Purchase Order
      When customer submits po
      Then po is saved in database
      And customer is notified

Scenario: Accepting a Purchase Order
      When clerk accepts Purchase Order
      Then Purchase Order is accepted
```

A Gherkin file uses `.feature` file extension. Every file starts with the `Feature` keyword. It is followed by a text which is the name of the feature and the lines below them is the description or narrative. Any text can be provided in the description except a line starting with one of the keywords used in the language. Gherkin keywords are as follows.

Feature	**Background**	**Scenario**	**Given**
When	**Then**	**And**	**But**
*****	**Scenario Outline**	**Examples**	

Here, the main keywords we would cover are Background, Scenario, Scenario Outline and Examples. The keywords Given, And, But, When and Then are used to start a step within the other keywords. The behaviour of the application is described in a feature file using the *scenarios*. Each scenario provides a concrete example of how the application should respond in a particular situation. Adding together all the scenarios would describe the expected behaviour of the feature. In Gherkin, we use mainly `Given`, `When` and `Then` keywords to identify different parts of a scenario. The `Given` keyword set up the context where the scenario is executed, `When` to start interacting with the system and `Then` to check if the expected result is the same as the outcome of the interaction. `And` and `But` keywords are used to add more steps to the above three keywords.

A simple example of a scenario in a Gherkin feature file is given below.

```
Scenario: Accept plant request
      Given the plant status is 'PENDING'
      When customer receives request to accept
      And customer accepts the plant
      Then status becomes 'ACCEPTED'
```

In the above scenario, we can see that the `And` keyword is used to add an additional `When` step. Similarly, we can use the `But` keyword to extend the functionalities of each of the three step keywords.

The *background* section provides a set of steps common to each scenario in the file. We use them in order to avoid repetition of steps in each scenario. Let us consider the following example where we define the initial database before the execution of each scenario.

```
Scenario: Accept plant request
      Given the following plants
            | _id | name       | description          | price  | status    |
            |  1  | Plant1     | Excavator 1.5 tons | 100.00 | PENDING   |
      When customer accepts the plant with _id '1'
      Then status becomes 'ACCEPTED'
Scenario: Reject plant request
      Given the following plants
            | _id | name       | description          | price  | status    |
            |  1  | Plant1     | Excavator 1.5 tons | 100.00 | PENDING   |
      When customer rejects the plant with _id '1'
      Then status becomes 'REJECTED'
```

We can see that the same database is initialized in multiple scenarios. The same can be rewritten using Background section as follows.

```
Background: Initialize plant catalog
      Given the following plants
            | _id | name       | description          | price  | status    |
            |  1  | Plant1     | Excavator 1.5 tons | 100.00 | PENDING   |

Scenario: Accept plant request
      When customer accepts the plant with _id '1'
      Then status becomes 'ACCEPTED'
Scenario: Reject plant request
      When customer rejects the plant with _id '1'
      Then status becomes 'REJECTED'
```

Scenario Outlines are used in those cases where there are multiple scenarios that follow the same pattern of steps and differ only in the input and expected values. The syntax of a scenario outline is similar to scenario except that scenario outlines have additional examples sections which provide the real values used in the steps. We use *placeholders* to represent the real values and substitute the values in the place of these placeholders from the examples defined in Scenario Outlines. Consider the following two scenarios with similar structure.

```
Scenario: Accept plant request
      When customer calls 'accept' on plant with _id '1'
      Then status becomes 'ACCEPTED'
Scenario: Reject plant request
      When customer calls 'reject' on plant with _id '1'
      Then status becomes 'REJECTED'
```

They can be made into a single Scenario Outlines as shown below.

```
Scenario Outline: Process plant request
      When customer calls <method> on plant with _id <id>
      Then status becomes <status>
      Examples:
            | method    | id  | status    |
            | accept    | 1   | ACCEPTED  |
            | reject    | 1   | REJECTED  |
```

3 Related Work

Although the adoption of the REST architectural style has been growing, the number of research papers related to the testing of the same is still very less. This section summarizes the existing testing methods of RESTful applications using the classic way of manual testing and using the automatic generation of test cases. The most common tools used to test an application programmed in Java are the frameworks like JUnit, NUnit and other xUnit frameworks. When they are used for testing web services they are tightly coupled with the implementation language under test [2]. As a result, developers are searching for methods to improve testing such services. Before we look into the various tools for test generation, let us look into the various approaches for the generation of tests.

Code based test generation is one of the most common ways to generate test cases. Several methods exist that facilitates in automatic generation of test cases based on the source code of application under test. Most of these methods concentrate on path and branch coverage of the code. These methods do not consider any model or specification. As a result, the tests cannot generate the expected output for the generated inputs.

Saswat Anand et al. [11] performed an orchestrated survey on the following automated test case generation methods.

- **Adaptive random testing**: This approach has been proposed as an extended random testing method. It has been found that the failure causing inputs tend to form a failure region. So non-failure inputs should correspondingly form a non-failure region. As a result, the new test case should be far away from the non-failure test cases when the previous test case is not a failure. Therefore, adaptive random testing enforces an even spread of randomly generated test cases across the input domain to enhance the failure detection effectiveness of random testing.
  Compared to random testing, even though this approach provides better test cases, it would require more memory and computation time because of the additional task of even spreading across the domain. So it does not necessarily have a better cost-effectiveness compared to random testing. Although adaptive random testing has more potential for cost-effective by reducing the time and space complexities.
- **Combinatorial testing**: It involves the process of selecting a sample of input attributes that complies with a subset of element combinations to be tested. The attributes and their input values are modelled as sets of factors and values. Using this model, the test cases are generated by selecting the Cartesian product of values of all factors. The overall testing cost of combinatorial testing is comparatively low and they facilitate higher code coverage. Higher efficiency combinatorial testing can be obtained only by using a modelling language. In such cases, the testing approach would require high skill level.
- **Search-based testing**: In order to maximize the achievement of test objectives by minimizing the test cost, search optimization algorithms, using some fitness function, are used to generate the test data. The primary concern of search-based testing is to generate a fitness function to capture the test objectives.
  A lot of research is going on over search-based testing recently. The test objectives that are measurable can be converted to a fitness function. Since any test objective can be converted to fitness function in principle, this approach is highly generic and widely acceptable. But there exist many other important test objectives which are not applicable for search-based testing yet. Also, the approach does not support multiple test objectives handling while searching for a test suite using a fitness function.

- **Symbolic execution**: It is a program analysis method, which analyses a program code and generates test inputs automatically for the program. It uses symbolic values as program inputs, instead of concrete values, and depicts the values of program variables as the symbolic expression of these inputs.
  The application of symbolic execution testing on real-world problems is limited dues to lack of research. However, it is used with other techniques like search-based testing to provide a better test case. Symbolic execution mechanism for test generation requires the source code. As a result, it cannot be used for Test-Driven development.
- **Model-based testing**: It is a lightweight method that uses models of the application systems to generate test cases. It generally puts the focus on behavioural testing. The models involved for the approach can either be in textual or graphical modelling format.
  This approach does not require the application code to generate tests. The maintenance effort is very low as the test plan is the model itself. It also ensures higher code coverage.

We have structured existing research and tools related to our proposed approach as shown in Figure 3.1, where we have organised the discussion into the generation of three main categories: Documentation, API Mock and Test cases.

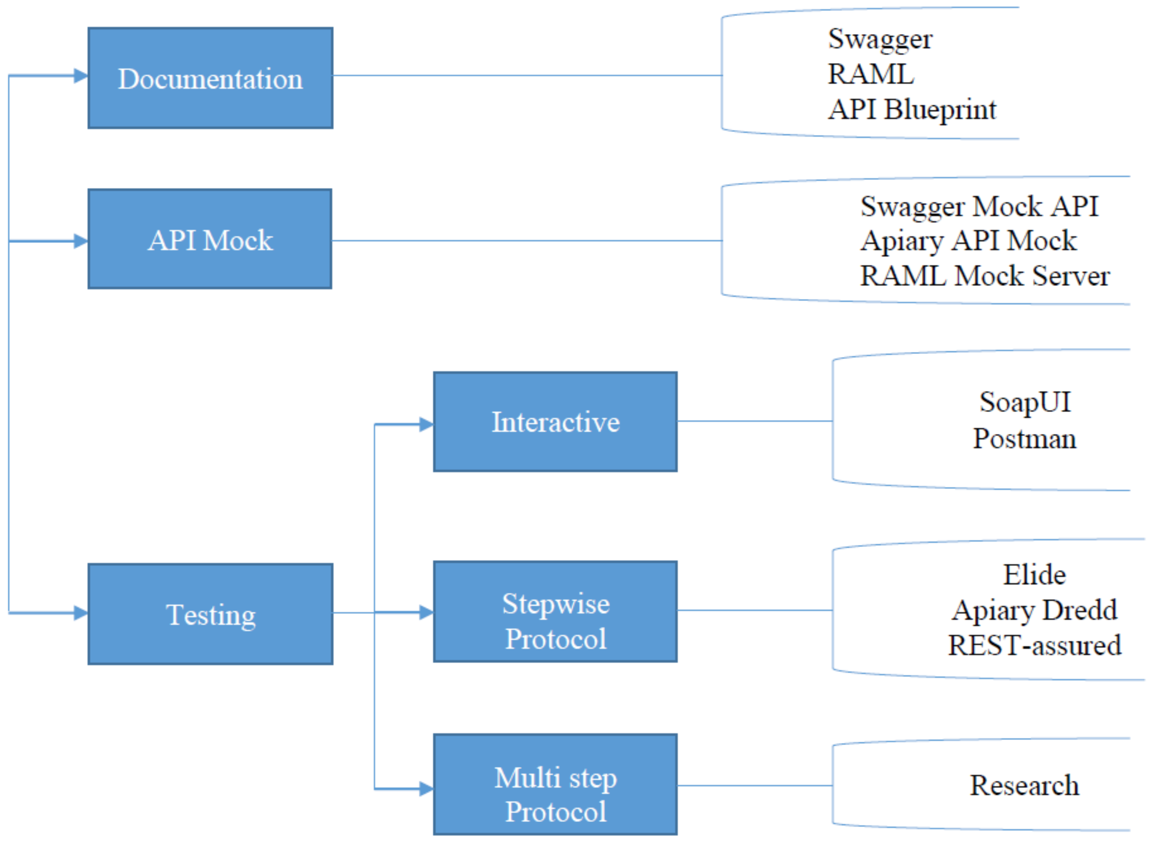


Figure 3.1 Related tools and research

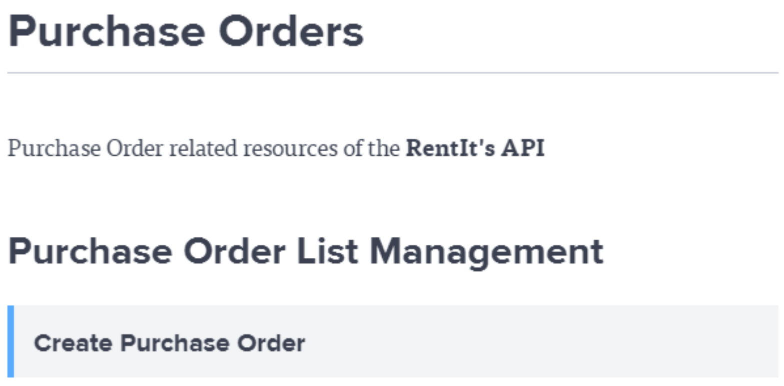
Now, let us discuss the related work among these three categories.

3.1 Documentation

The documentation is just the generation of a document which can be used as a contract between the server and client systems in order to let the clients access their application. The documentation specifies a well-formatted document what could be the input to a scenario and what the output should be like. This section describes the various tools that were identified to generate the documentation of a REST API.

One very common open source tool is Swagger[1] which can be used to document and describe RESTful APIs. The document specification defines a set of files which are used to describe such an API. These files are used by other tools under swagger such as Swagger-Codegen[2] for generating the server implementation in different languages and Swagger-UI[3] for displaying the API. Swagger support is incorporated in many commercial tools in the market such as Microsoft Visual Studio, vREST etc.

RESTful API Modelling Language (RAML)[4] is another tool used for documenting a REST API. It is a tool which is becoming very common these days and it is being used by hundreds of open source tools like API Console, API Notebook etc. to create specific custom documentation. Both Swagger and RAML documentations make use of YAML (recursive acronym for YAML Ain't Markup Language) as the documentation language. YAML is a human-readable language which takes its concepts from other programming languages like Perl, Python and XML.


Purchase Orders

Purchase Order related resources of the **RentIt's API**

Purchase Order List Management

Create Purchase Order

Figure 3.2 API document for Creation of Purchase Order

API Blueprint[5] helps in generating high-level API document for web APIs. The blueprint is a Markdown document that can be used to describe REST API. It is an extension of markdown language tailored to provide the details of the interaction with the API. The document is structured into specific logical sections, with each section specifying distinctive meaning. It comes along with many tools that could mock and test the API. It also provides another tool called RSpec API Blueprint[6] that facilitates the generation of API documentation by making use of the API blueprint format from request specs which can be used for testing the

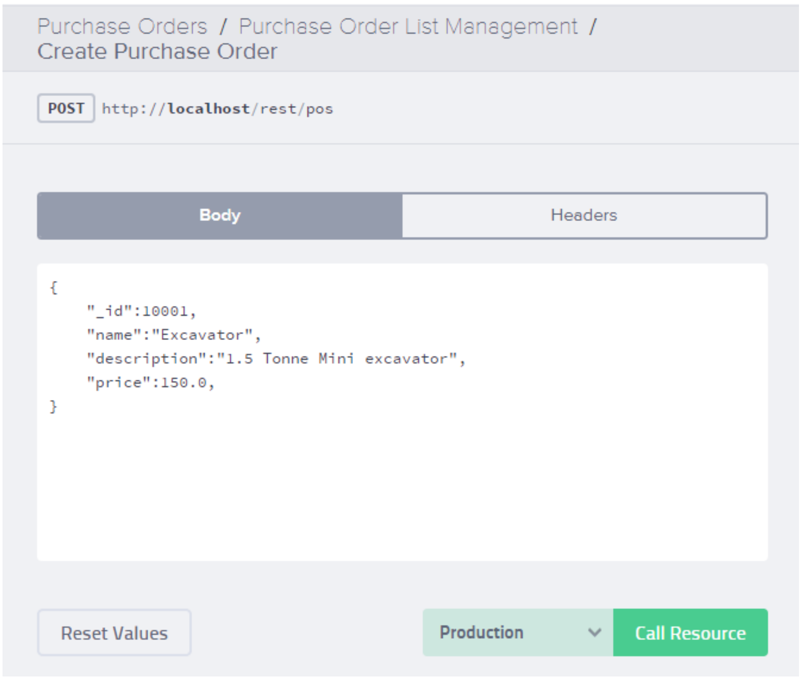
[1] http://swagger.io/specification/
[2] http://swagger.io/swagger-codegen/
[3] http://swagger.io/swagger-ui/
[4] http://raml.org/
[5] https://apiblueprint.org/
[6] https://github.com/calderalabs/rspec_api_blueprint

REST APIs. An example of an API Blueprint document with a single scenario of the creation of Purchase Order is provided in Figure 3.2.

One drawback about the tools above is that the generated document is very verbose. It requires a lot of lines to generate a properly structured document. Also, the documentation is more technical when the tools use API blueprint and YAML languages.

3.2 API Mock

Mocking an API is the process of creating and using a replacement version of the API instead of using an actual software API. It will behave as the actual API, but it will lack many functional and non-functional logic of the actual API. This section discusses various tools we came across that can be used to generate mock API.



http://localhost/rest/pos

```
{
    "_id":10001,
    "name":"Excavator",
    "description":"1.5 Tonne Mini excavator",
    "price":150.0,
}
```

Figure 3.3 API Mock of Creation of Purchase Order

A very common tool used for mock server generation from the API specification is the apiary's API-Mock[7]. The user has to document the REST API in API blueprint format and the tool generates a simple and fast mock server using the routes in the document and provides the responses defined in the API specification. An example of API mock for the creation of Purchase Order scenario is provided in Figure 3.3. The request body can be seen in the figure. When the user clicks on "Call Resource" button, the REST call is made to the mock server method running at `http://localhost/rest/pos` and returns the static response provided in the document.

Swagger Mock API[8] is an npm module. It is a connect-compatible middleware used in generating functions, which in turn generates a REST API based on Swagger compatible YAML or JSON file. The Osprey Mock Service[9] is another npm module used for mocking

[7] https://github.com/localmed/api-mock
[8] https://www.npmjs.com/package/swagger-mock-api
[9] https://github.com/mulesoft-labs/osprey-mock-service

the services from RAML definition. Again, the execution of Swagger and RAML mock service is similar to the apiary. They make use of examples to provide the static responses. All these tools respond with fixed data instead of dynamic data. Also, in order to set up a complete mock server of an API, it would require a lot of documentation work.

3.3 Test cases

Most of the research regarding the related work was concentrated on the test case generation. We have divided the identified tools and researches related to automatic test case generation into three sections. Interactive testing tools are the tools which can be used to test against a published REST API. Stepwise protocol testing tools are those tools which could perform a single functionality testing of a REST API. Multi-step protocol testing involves the testing tools and research which aimed at automatic test generation where the tests could test multiple steps at a time. The tools and research in the three sections are briefly described below.

2.4.1 Interactive Testing

The cases we consider in interactive testing are all tools for script-based testing, which is the process of testing inputs and observations programmed in scripts using some dedicated or general-purpose languages. Users are provided with an application by means of which they can provide sample requests, which could include the HTTP verb, HTTP headers and the test data, and check out the output of the underlying REST application. They can record a sequence of calls in the form of a test script (e.g. a test workflow by some vendor) that can be later used to test typical sequences of calls in the form of scenarios.

One such approach is the SoapUI[10] tool which uses a Service-Oriented-Architecture to configure functional, compliance, regression and load tests for web-services. SoapUI provides test coverage and supports most of the technologies and standard protocols.

Postman is a script-based testing tool which comes as a Chrome browser plug-in. It is used to test a REST API. It has a beautiful user interface for entering parameters. It does not require the user to deal with commands. It also helps the user to automate the process of making API requests and testing API responses.

2.4.2 Stepwise Protocol Testing

Apiary's Dredd[11] is a tool which can be used for testing API documentation described using API Blueprint against its backend implementation. It helps to have the RESTful API documentation up-to-date by plugging the documentation to other continuous integration systems.

Haleby's REST-assured[12] is another approach to improving the test case development of RESTful services. Haleby developed a tool for rapidly writing test cases for any RESTful application, which worked on the when-then rule. Every test cases could be configured using the fluent interface. As a result, the test cases were concise and efficient.

Elide[13] is a framework which uses JSON API to develop RESTful applications. It has a sibling framework which is used to specify test cases for the various CRUD services of Elide. The framework uses Gherkin feature file as the domain specific language. A list of

[10] http://www.soapui.org/
[11] https://dredd.readthedocs.org/
[12] https://github.com/jayway/rest-assured
[13] http://elide.io/

collection and entities used by the application is specified in the gherkin file and the framework tests all their combinations of CRUD operations by checking the actual result from Elide with the expected result defined in the gherkin file. Any mismatches are considered as test failures.

2.4.3 Multistep Protocol Testing

Chakrabarti and Rodriquez [12] introduces an algorithm which tests whether a service conforms to the property of connectedness of REST. Connectedness means that all other resources of a service are accessible from a root resource. Klein and Namjoshi [13] in the same context, formalize the properties and concepts of REST. This could be used to check the behaviour of the system under test. These two methods verify the REST constraints, former by using graph models and the latter by using a formal specification of the system under test.

Another approach that uses automated test case generation is called Test-The-Rest (TTR) [14] and a tool was created implementing this approach with the same name. It used XML notations to configure each test case, which also had the facility to write test cases which contained a sequence of other test cases. Each test case has pieces of information like the HTTP method, resource URI and the expected representation. But the approach had to manually configure each test cases in the XML to generate them which was a tedious task.

Fertig and Braun proposes a method for automated test case generation by using Model Driven Testing (MDT). They were able to generate test cases based on the model, which only contained the design of the API under test and had no particular information regarding testing. The approach used textual Domain Specific Languages (DSLs) to depict the model used for test generation.

Pinheiro, Endo and Simao proposes a Model-Based Testing (MBT) approach [15] which promotes behavioural testing of RESTful APIs, providing a more formal and systematic testing. They chose UML protocol state machine as the model since it gives importance to the transition between the states and not the action that occurs in each state and thereby providing the required level of abstraction. In other words, it focusses on the effect of transition than the behaviour of the states. The approach was implemented using a tool which was developed in Java that generated test suites for state and transition coverage.

3.4 Discussion

The documentation of the API is generally used by the client systems to access the server machine. The tools discussed above would require a large number of lines of code in order to create a well-formatted document. In our approach, we consider a Gherkin feature file as the documentation used as Gherkin by itself is an intuitive and easy to understand the document written using plain spoken language which can be understood by the server side and end users.

The mock servers generated by the tools discussed above mostly use examples to communicate which in turn consider only a single scenario at a time. They are designed in such a way that they could execute only one functionality, say “Creation of a Purchase Order”. Their result is static as they provide static responses. Our proposed tool generates a mock from the documentation (Gherkin file) which not only aims to consider a single scenario execution, but additionally, aims to provide a functionality of chaining by which we could execute multiple scenarios. Also, it provides dynamic data, which will be the actual response when a request is sent to the mock.

A research was done on the interactive testing tools. But they are used to test published APIs. Since we concentrate more with the automatic generation of test cases and the mock controller generation and do not deal with the actual API and its testing, we content that the tools are out of scope for our approach. The test cases generated by single protocol testing tools mainly concentrate on the execution of a single scenario. Our approach generates test cases, which are tested with the mock controller by default. Also, they are able to consider a single scenario as well as multiple scenarios in a single unit test case.

The multistep protocol testing researches mainly require some models and the target application. If we map these researches to a state model, it would execute based on a sequence of transitions. But this would imply multiple completely independent test cases in a single test case, instead of keeping the state of the previous test case and providing a chaining of the scenarios. Our method focuses on chaining by testing multiple scenarios, keeping the state alive from the previous scenario. The aim of our method is to guide the developers to follow a TDD approach in order to implement the target application.

4 Method

This chapter provides an overview of the architecture that was used for the implementation and the corresponding approach taken to implement the DSL and its transformation to Java code. But in order to generate Java code, we could make use of simple UML modelling diagrams instead of a DSL. The reason why we have chosen DSL over UML diagrams is that the diagrams are more compact with the amount of information it could provide. By combining multiple UML diagrams, we could obtain information regarding the application, yet it is a tedious task. This is why we have chosen to use DSLs as they are more direct, appealing and easier to use [16] and they could be provided with as much details of the application as possible.

The next discussion was to decide the structure of a DSL as it should be able to provide all the relevant low-level information of the application in order to generate the complete mock controller methods. Initially, we decided to create a DSL structure providing information of each method, but then the idea was dropped and chose Gherkin language as the DSL. The reason behind using Gherkin is that it is very widely used among software development processes and that it can be written in plain spoken language which can be understood by all users. Also, a Gherkin file could be used to evaluate the behaviour of the application generated using TDD approach from the test cases generated using our method. The native Gherkin file was custom tailored for our purpose by providing the low-level details in them like the method name, the Uri, the verb etc.

Now that we have decided the DSL structure, we need a platform to implement a grammar for the Gherkin language DSL. Since the targeted language is Java, we searched tools for implementing DSL associated with Java. The main criteria we chose to select the tool were the following.

1. It should have support to implement the grammar of the language.
2. It should have support for transforming one representation to another one.
3. It should contain a textual editor for the DSL so that the users could write code to translate the DSL to the desired code.

Considering the above requirements, it was decided that Xtext would be the best option. Additionally, Xtext came with Eclipse IDE which had support for Spring MVC framework. Moreover, it is open source and supported by the Xtext community. Recently, the community upgraded Xtext to be compatible with IntelliJ IDEA, although we did not proceed our implementation to that area.

In the following sections, we provide a brief introduction to the architecture of Xtext followed by the architecture of our proposed solution.

4.1 Xtext

Xtext is a very sophisticated tool which helps to implement simple domain specific languages with IDE support. If a language lacks decent tool support, Xtext[14] can be used to provide an Eclipse-based development environment. This environment is responsible to provide editing experience from modern Java IDEs in a short amount of time. The compiler components of the language can be used in any Java environment. Unlike XML, which has a strict syntax, there is no limitation in Xtext regarding syntax. It helps to define the required

[14] https://eclipse.org/Xtext/

syntax. The task of reading the model, parsing or working with it and writing it back is simplified to a greater extent using Xtext. Figure 4.1 shows the architecture of Xtext.

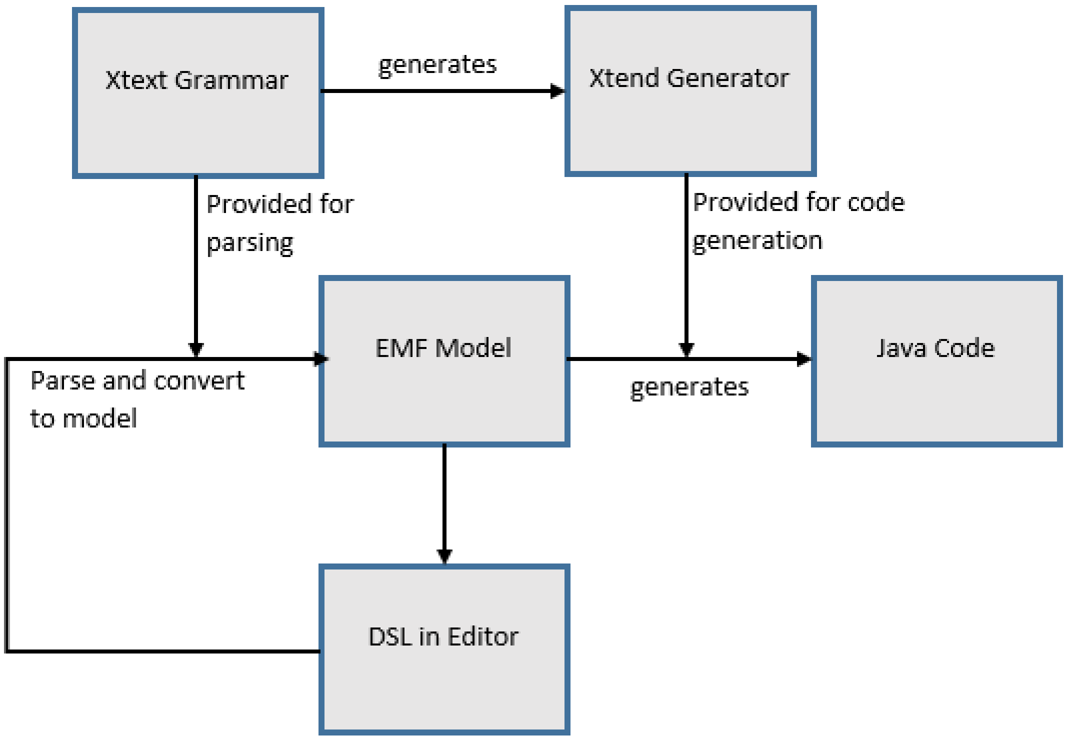


Figure 4.1: Xtext architecture

Xtext provides a grammar definition language from which it generates a runtime support for the language in connection with metamodel in order to apply the lexical parsing. Xtext uses Eclipse Modelling Framework[15] (EMF) models as the in-memory notation of any parsed file. It decorates the model by using syntax highlighting rules and additionally allows us to specify code generation support. This code generation is specified in another language provided by Xtext called Xtend[16]. It would have been possible to generate the code using Java, but the number of lines of code was very much higher compared to what the lines of code in Xtend.

Xtend is a statically-type programming language used along with Xtext in order to translate the Xtext grammar to Java source code. It is based on Java programming language. With Java, it has zero interoperability issues. Every code written in Xtend interacts directly with Java as expected. Xtend also provides additional features like call-hierarchies, debugging, rename refactoring etc. The EMF code generator will be run on the Xtend generator to generate the desired Java code.

[15] http://www.eclipse.org/modeling/emf/
[16] http://www.eclipse.org/xtend/

4.2 Architecture overview

The architecture of our proposed approach is illustrated in Figure 4.2.

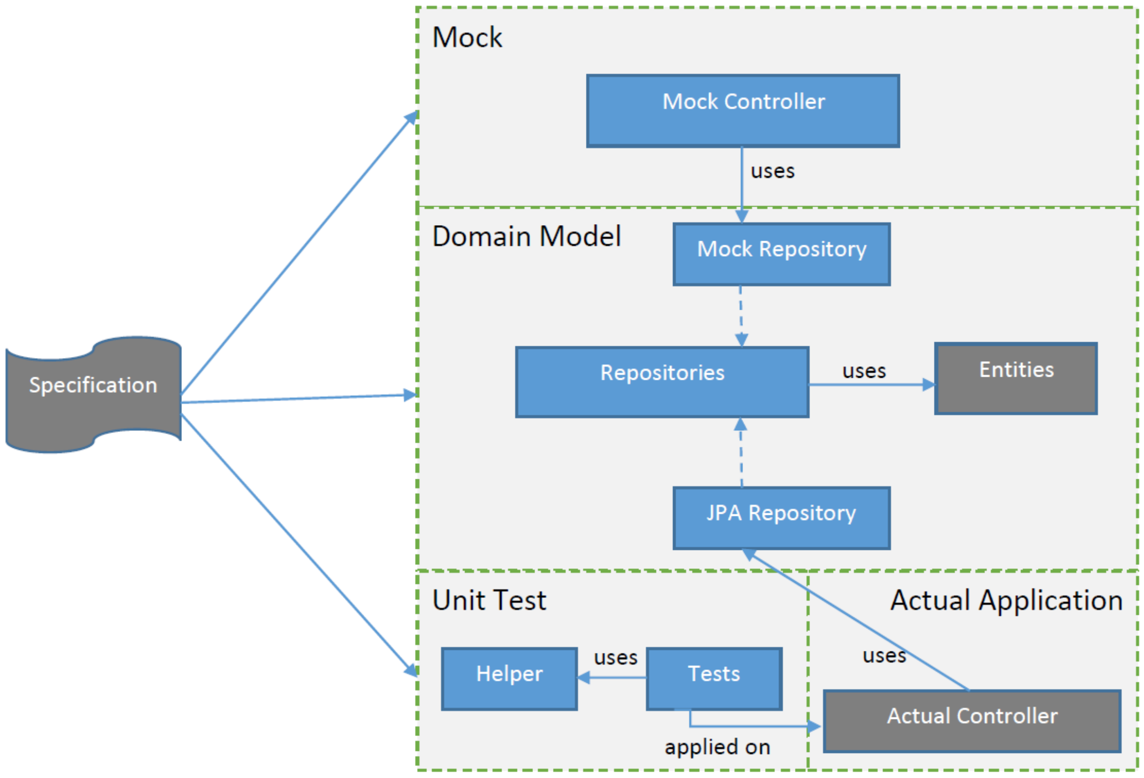


Figure 4.2: Process of Mock controller and Test generation

The modules Specification, Entities and the Actual controller (which highlighted in grey colour) should be provided by the user and the ones in blue are generated. The main part of the code generation is the specification DSL which must be provided by the user. As discussed before, we consider Gherkin language as the DSL so that it could cover resource model as well as the state model. The user has to provide the specification followed by the entities used in the application. The entities should be created in such a way that it supports both Mock and JPA repositories. The actual controller is supposed to be developed by the user only after the tests are generated using TDD approach.

Using the specification, the mock repository is generated first followed by the JPA repository. The mock repository will be used by the mock controller for performing basic database operations. They will only support the basic CRUD operations. The mock controller is also generated using each scenario in the specification. The tests generated are first tested on the mock controller and verified before they are used to create the actual controller using TDD.

After the generation of mock, we turn to the generation of the tests. The test data is to be taken from the specification. Instead of loading all test data in the main unit test class, we generate a helper class which stores all the data from the specification used for testing. Finally, the test class is generated covering all the scenarios provided in the specification. Once it is generated, the test cases are verified against the mock controller. If they are successful, then the actual controller is supposed to be developed using the test cases one by one. The complete application is ready once all the tests pass in the actual controller.

5 Code Generation

We consider the scenario, originally described by Dumas in [17], where an equipment rental company called RentIT rents out a wide range of construction equipment. A shortened version of the scenario covering the workflow from creating and order to closing the order is provided in Appendix A.

The discussion regarding the implementation will be based on the state transitions a Purchase Order (PO) undergoes during an equipment rental process in RentIT. All the scenarios can be referred from Appendix A. The complete transitions are illustrated by the state chart below.

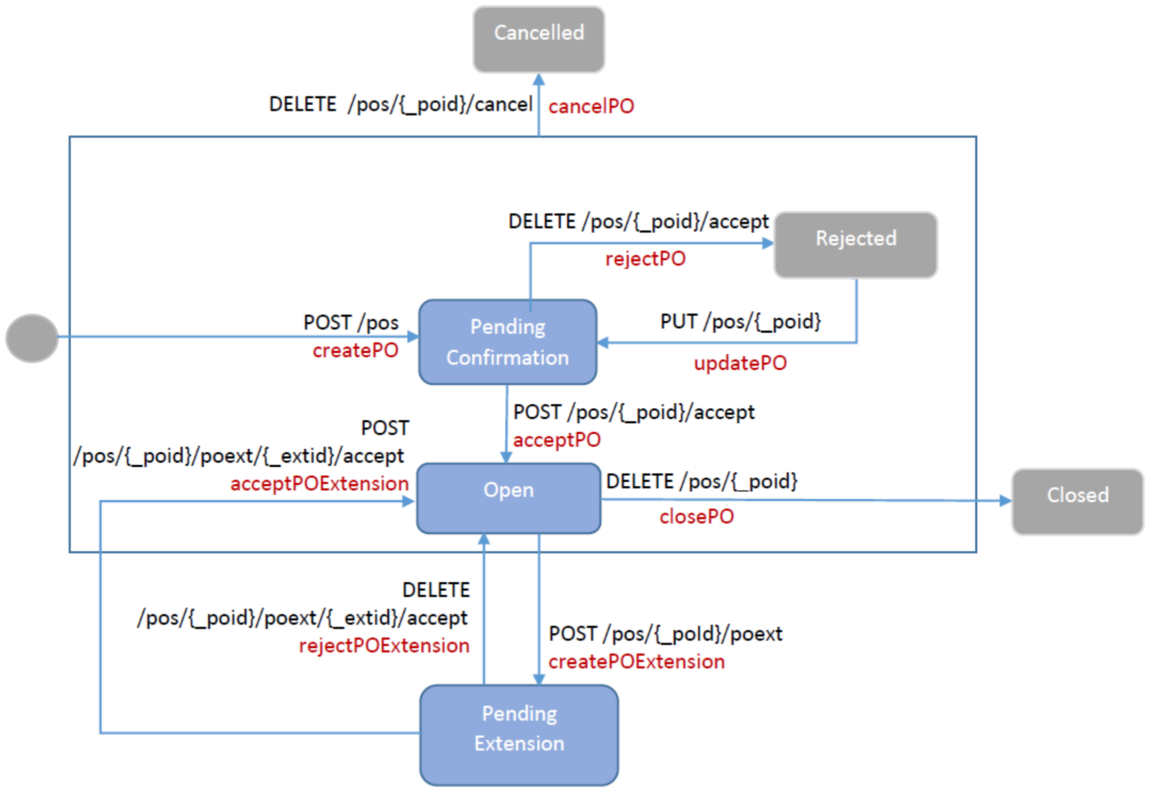


Figure 5.1 State transition diagram of Purchase Order

The list of methods and their verb and Uri inferred from Figure 5.1, which provides the complete state transition diagram of the application, are provided in Table 5.1.

Method name	**Verb**	**Uri**
createPO	POST	/pos
acceptPO	POST	/pos/{_poid}/accept
rejectPO	DELETE	/pos/{_poid}/accept
updatePO	PUT	/pos/{_poid}

closePO	DELETE	/pos/{_poid}
cancelPO	DELETE	/pos/{_poid}/cancel
createPOExtension	POST	/pos/{_poid}/poext
acceptPOExtension	POST	/pos/{_poid}/poext/{_extid}/accept
rejectPOExtension	DELETE	/pos/{_poid}/poext/{_extid}/accept

Table 5.1 Methods involving Purchase Order state transitions

In order to begin the implementation, we consider a small section of the complete scenario. The rental process starts when a new Purchase Order (PO) is received by the rental company from a customer. A PO will have an inventory item along with the start date and end date which marks the rental period. When a Purchase Order is received, the sales representative of RentIT verifies the PO and checks the availability of the item requested in the PO. This can lead to one of the following 2 outcomes.

1. the PO is accepted
2. the PO is rejected

The implementation phase is started with the above simple workflow of a Purchase Order. The corresponding state transition diagram we consider for the discussion is provided in Figure 5.2.

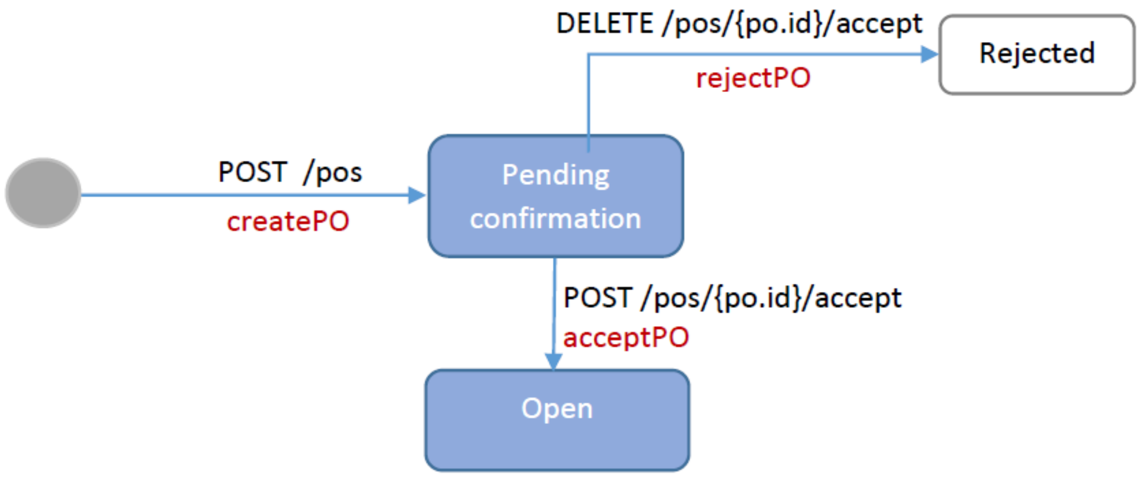


Figure 5.2 State transition diagram of Purchase Order considered for discussion

First, the grammar is defined for the Gherkin structure as we are using a Gherkin feature file to generate the test cases. The mock is generated first from the feature file. The mock will contain all the methods defined in the feature file. Then a helper file to provide input data to the test cases is generated. And finally, the test cases are generated from the feature file.

5.1 Specification of DSL

A feature file always starts with the keyword “Feature” followed by a text which becomes the title of the feature. In our case, we call it “PurchaseOrder feature” as we are dealing with Purchase Order scenarios. It is followed by a narrative. The narrative is optional as we do not consider the narrative for code generation.

```
Feature: PurchaseOrder feature
      As a customer
      In order to rent plant equipment
      I need to process a Purchase Order
```

Now let us consider the first action associated with the rental system. A request is received from the client for the creation of a Purchase Order. A high-level Gherkin scenario for this action can be defined as follows.

```
Scenario: Creation of Purchase Order
      When customer submits po
      Then po is saved in database
      And customer is notified
```

Here, we can see a step which says *po is saved in database*. But we have not defined anything as the database yet. Therefore, before considering the different scenarios of Purchase Order, we need to setup the initial database. It is optional for simple APIs, but in our case study, it is necessary to have at least some Plant objects in the database in order to create Purchase Orders. So we define a Background which provides the initial database structure.

```
Background: Initial plant catalog and purchase orders
  Given the following plants
    | _id | name           | description        | price  |
    |  1  | Mini excavator | Excavator 1.5 tons | 100.00 |
    |  2  | Mini excavator | Excavator 2.5 tons | 120.00 |
    |  3  | Midi excavator | Excavator 3.0 tons | 150.00 |
    |  4  | Maxi excavator | Excavator 5.0 tons | 200.00 |
  Given the following purchase orders
    | _id | plant           | startDate  | endDate    | cost    | poStatus |
    |  1  | {"_id": 1, …}¹ | 2016-02-29 | 2016-03-19 | 2000.00 | PENDING  |

   1. { "_id":1, "name":"Mini excavator", "description":"Excavator 1.5 tons", "price":100.0}
```

Due to lack of space, the lengthier data is provided as a footer information as shown above under plant column.

In the above case, we could provide some notation to identify the databases corresponding to Plant and Purchase Order, say *$Plants* and *$PurchaseOrders* respectively, so that while parsing the feature file, we could identify them as databases. This notation is selected because there is a technical reason which we would introduce later. So getting a Plant object from the database with id=1 can be denoted by *$Plants.findOne(1L)*. We also define a method to convert from any object to JSON string. We could provide the notation *$toJson* for this object. We adapt these notations so that we could identify them while parsing using some parser during code generation phase. So the final Background can be rewritten as follows.

```
Background: Initial plant catalog and purchase orders
  Given the following $Plants
    | _id | name           | description        | price  |
    |  1  | Mini excavator | Excavator 1.5 tons | 100.00 |
    |  2  | Mini excavator | Excavator 2.5 tons | 120.00 |
    |  3  | Midi excavator | Excavator 3.0 tons | 150.00 |
    |  4  | Maxi excavator | Excavator 5.0 tons | 200.00 |
  Given the following $PurchaseOrders
    | _id | plant          | startDate  | endDate    | cost    | poStatus |
    |  1  | {"_id": 1, …}[1] | 2016-02-29 | 2016-03-19 | 2000.00 | PENDING  |

   1. #{$toJson($Plants.findOne(1l))}
```

Now let us consider the scenario of creation of Purchase Order once again.

```
Scenario: Creation of Purchase Order
      When customer submits po
      Then po is saved in database
      And customer is notified
```

The above scenario is too abstract. It does not provide any technical information like what the value for po is, how the customer submits the po or how the customer is notified. Taking these technical aspects also into consideration, we can reformat the scenario as follows.

```
Scenario: Creation of Purchase Order
      When customer calls 'createPO' using 'POST' on '/pos' with 'po'
      Then PurchaseOrders must contain 'po'
      And status code must be '201'
      And location must have '/pos/poId'
      And po status must be 'PENDING'
```

This provides low-level details like the method name (`createPO`), the uri to the method (`/pos`), the parameter name (`po`) etc. Once the execution of the method is done, the PurchaseOrder table should contain the newly created PurchaseOrder object (`po`). Using status code of 201, we can inform the successful creation of PO to the client. Also, the location of the created PurchaseOrder will be `/pos/poId`, where `poId` will be the id of the created object. An additional step is added to check if the Purchase Order status has been updated to PENDING.

The above scenario can be further improved by providing the database notation for PurchaseOrder used in the background (*$PurchaseOrders*) for consistency. Also, in order to understand that `poId` is the id of the created `po`, we can replace `poId` by `po._id` assuming that the id of a Purchase Order is named as *_id*. Finally, we provide a variable name for Purchase Order status as `poStatus`, which should be the name provided for the attribute in the model. So the scenario can be rewritten as follows.

```
Scenario: Creation of Purchase Order
      When customer calls 'createPO' using 'POST' on '/pos' with 'po'
      Then $PurchaseOrders must contain 'po'
      And status code must be '201'
      And location must have '/pos/<po._id>'
      And 'poStatus' must be 'PENDING'
```

Even with the above scenario, one cannot understand what the value for `po` is. But for the time being, we will consider the next action and postpone improving this scenario for later. Once the Purchase Order is created, if a plant is available, then the PO is accepted, else it is rejected. Ideally, this functionality should happen automatically. But since it requires a lot of business logic, which would make our feature long and dirty, we would avoid automating this functionality. Instead, we consider them as manual tasks by some actor. So there are two actions now and the actor is the clerk. We will consider acceptance of PO first. The corresponding scenario is given below.

```
Scenario: Accepting a Purchase Order
      When clerk accepts Purchase Order
      Then Purchase Order is accepted
```

Again this is a high-level scenario. We reformat them like the way we did while writing the scenario for the creation of PO.

```
Scenario: Accepting a Purchase Order
      When clerk calls 'acceptPO' using 'POST' on '/pos/{id}/accept'
      Then 'po' should be '$PurchaseOrders.findOne(id)'
      And status code must be '200'
      And 'poStatus' must be 'OPEN'
```

Similarly, for rejection of a Purchase Order, the scenario will be as follows.

```
Scenario: Rejecting a Purchase Order
      When clerk calls 'rejectPO' using 'DELETE' on '/pos/{id}/accept'
      Then 'po' should be '$PurchaseOrders.findOne(id)'
      And status code must be '200'
      And 'poStatus' must be 'REJECTED'
```

The above two scenarios are almost a replica of each other except the field values. Therefore, we could use a Scenario Outline instead of a Scenario and use placeholders instead of the field values. Thus, the above two scenarios can be combined as given below.

```
Scenario Outline: Processing of a Pending Purchase Order
      When clerk calls <function_name> using <verb> on <uri>
      Then 'po' should be '$PurchaseOrders.findOne(id)'
      And status code must be <status>
      And 'poStatus' must be <poStatus>
      Examples:
        | function_name | verb   | uri              | poStatus | status |
        | acceptPO      | POST   | /pos/{id}/accept | OPEN     | 200    |
        | rejectPO      | DELETE | /pos/{id}/accept | REJECTED | 200    |
```

Similarly, we can rewrite the scenario for "Creation of Purchase Order" also as Scenario Outline. The `po` also could be made into a placeholder depicting the JSON value of the `po` object. Also, we can make use of JSON Patch and JSON Merge Patch in order to update a JSON by sending the changes rather than the whole new JSON. In our case, we can use JSON Patch to change one attribute and JSON Merge Patch when multiple attribute values have to be changed. Since, in the above case, we are only updating the `poStatus` from PENDING to OPEN/REJECTED. So we could use JSON Patch. In the case of "Creation of Purchase Order", we are updating the `poStatus` and the `cost`. So we could use JSON

Merge Patch. We would adopt some notation for them like the *$toJson* notation for a method to convert to JSON in Background section. We make use of *$patch(<JSONObject>,<PatchString>)* and *$mergePatch(<JSONObject>,<MergePatchString>)* to represent JSON Patch and JSON Merge Patch respectively. So all the scenarios can be finally written as follows.

```
Scenario Outline: Creation of PurchaseOrder
  When customer calls 'createPO' using 'POST' on <uri> with <po>
  Then $PurchaseOrders must contain $mergePatch(<po>,<po_patch_merge>)
  And status code must be <status>
  And location must have <location>
  And 'poStatus' must be <poStatus>
  Examples:
  | uri   | po          | status | location       | po_patch_merge    | poStatus |
  | /pos  | {"plant": …}[1] | 201    | /pos/<po._id>  | {"id": …}[2]          | PENDING  |

Scenario Outline: Processing of Pending PurchaseOrder
  When clerk calls <function_name> using <verb> on <uri>
  Then <po> should be '#{$PurchaseOrders.findOne(id)}'
  And $PurchaseOrders must contain $patch(<po>,<po_patch>)
  And status code must be <status>
  And 'poStatus' must be <poStatus>
  Examples:
  | function_name| verb   |uri               | status| po_patch   |id| poStatus|
  | acceptPO     | POST   |/pos/{id}/accept| 200   | [{"op": …}][3] |1L| OPEN    |
  | rejectPO     | DELETE|/pos/{id}/accept| 200   | [{"op": …}][4] |1L| REJECTED|

   1. {"plant": #{$toJson($Plants.findOne(1l))}, "startDate": "2016-02-29", "endDate": "2016-03-04"}
   2. {"_id": #{$PurchaseOrders.count()+1}, "poStatus": "PENDING", "cost": "200"}}
   3. [{"op": "replace", "path": "/poStatus", "value": "OPEN"}]
   4. [{"op": "replace", "path": "/poStatus", "value": "REJECTED"}]
```

We need to generate the test cases using chaining of the Scenario Outlines. This can be facilitated by using scenarios. For example, the creation and acceptance of a Purchase Order can be denoted by the following.

```
Scenario: Create and accept PurchaseOrder
      When scenario "Creation of PurchaseOrder" with [1]
      And scenario "Processing of Pending PurchaseOrder" with [1]
```

Here, the number provided in the square brackets denotes the corresponding example number. In the case of the scenario "Creation of PurchaseOrder", there is only one example, while in the case of "Processing of Pending PurchaseOrder", [1] denotes the example for acceptPO and [2] for rejectPO.

The complete Gherkin feature file created during the discussion in this section is provided in Appendix B. Using this feature file, we proceed to implement the grammar.

5.2 Grammar definition

The actual grammar for a Gherkin feature file was obtained from [18].

When defining a grammar, the first and foremost task is to define a start entity that acts as the root node. As the grammar is dealing with a cucumber feature file, *Feature* is the name chosen for the starting entity.

The feature file usually starts with the keyword “Feature:” followed by a title. It can optionally contain one or more tags and a narrative description of what the feature does. In our context, we have avoided these sections as we do not require them for the code generation. The main sections of the file we consider are the sections provided below.

```
Feature:
      tags+=Tag*
      'Feature:'
      title=Title EOL+
      narrative=Narrative?
      background=Background?
      scenarios+=(Scenario | ScenarioOutline)+;

Background:
      'Background:'
      title=Title? EOL+
      narrative=Narrative?
      steps+=Step+;

Scenario:
      tags+=Tag*
      'Scenario:'
      title=Title EOL+
      narrative=Narrative?
      steps+=Step+;

ScenarioOutline:
      tags+=Tag*
      'Scenario Outline:'
      title=Title EOL+
      narrative=Narrative?
      steps+=Step+
      examples=Examples;

Step:
      stepKeyword=StepKeyword
      description=StepDescription EOL*
      tables+=Table*
      code=DocString?
      tables+=Table*;

Examples:
      'Examples:'
      title=Title? EOL+
      narrative=Narrative?
      table=Table;
```

A background is similar to a scenario just that it is invoked before every scenario is run. It starts with the keyword “Background:” followed by the title and then the steps. It can optionally have a description. The grammar for Background is provided above.

Scenario is the core of Gherkin structure. It starts with “Scenario:” keyword followed by the title. Each scenario can have one or more steps. Scenario outline is used to avoid repetition of scenarios for different values. It starts with the keyword “Scenario Outline:” followed by its title. Scenario outlines contain placeholders which are replaced by the values from the examples during their run. Apart from the additional examples in scenario outline, its structure is similar to that of a scenario. The grammar for Scenario and ScenarioOutline is given above. In the course of the thesis, ScenarioOutline is used to provide the various processes within the application (e.g. Creation of Purchase Order) and Scenario to provide the sequence of scenario outlines. The Background, Scenario and ScenarioOutline contain one or more steps. A step starts with a step keyword which can be one of the following: *Given, When, Then, And, But*. The step keyword is followed by a step description. The description can either be simple text or text containing placeholders (within <> in the case of ScenarioOutline). A step can optionally contain tables. The examples in ScenarioOutline usually contains a table to replace the values in the corresponding placeholders in the steps. The grammars for Step and Example is provided above. The complete grammar definition is presented in Appendix C.

5.3 Provision of Domain model

Before we start the implementation of the mock, it is required to have a project structure with the necessary models and other related classes or enums. These models have to be manually provided.

In our case, it is evident that we consider 2 models, namely PurchaseOrder and Plant. From the feature file, we can infer the various attributes of the models. Plant has the following attributes: `_id`, `name`, `description` and `price`. PurchaseOrder has the following attributes `_id`, `plant`, `startDate`, `endDate`, `cost` and `poStatus`. We consider the `poStatus` as an enum. So we need to create an enum for the `poStatus`.

So the initial structure of PurchaseOrder model will be like

```
public class PurchaseOrder {
      Long _id;
      Plant plant;
      LocalDate startDate;
      LocalDate endDate;
      Double cost;
      POStatus poStatus;
}
```

But in the real world scenario, these models are supposed to be saved in the databases. In order to facilitate database saving functionality, it is required to make some changes to the model. Firstly, we must provide some annotations for the *_id* which are

- org.springframework.data.annotation.Id
- javax.persistence.Id – to denote that this attribute is the primary key of the JPA entity.
- javax.persistence.GeneratedValue – to denote that the attribute value is automatically generated.

We use Spring HATEOS as a middleware to provide hypermedia support. We get this support by extending the class *org.springframework.hateoas.ResourceSupport.* So the second change to the model is to extend `ResourceSupport`. It is also advisable to provide *javax.persistence.Enumerated* annotation for enums used in a model (in our case `poStatus`). Finally, when a model references another model, the corresponding association type should also be annotated (*OneToOne* in the case of `plant` above). So the final structure of PurchaseOrder looks like follows.

```
public class PurchaseOrder extends ResourceSupport {
      @org.springframework.data.annotation.Id
      @Id @GeneratedValue
      Long _id;

      @OneToOne
      Plant plant;
      LocalDate startDate;
      LocalDate endDate;
      Double cost;
      @Enumerated(EnumType.STRING)
      POStatus poStatus;
}
```

Similarly, we should manually create the other models used in the application before the code generation. Once it is done, we carry forward to generate the mock. But before we deal with the mock, we should generate the corresponding repositories from the Cucumber feature for each model. It is because the mock controller will be using these repositories to find, save or delete based on the operations provided in the Cucumber feature and we need the repositories for these tasks. It is ideal to use two different repositories. For the mock, we generate the corresponding CrudRepository (*org.springframework.data.repository.CrudRepository*) and for the actual controller, we use the JpaRepository (*org.springframework.data.jpa.repository.JpaRepository*). The CrudRepository is used for the mock in order to implement CRUD operations without having an actual database. In other words, the database was mocked by using CrudRepository. The method of generation of repositories is provided in the following section.

5.4 Generation of Mock Controller

As mentioned above, before we begin the generation of the mock controller, it is required to generate the corresponding repositories. We create two repositories for a single model to be used by the mock controller and the actual controller. The only information needed to generate the repositories is the model name as the repositories we intend to generate are basic repositories without specific querying functionalities, which implies that the only different between the repositories will be the corresponding model name.

Now we need to figure out how we could obtain the models used. We had provided a Background section in the feature which provided information regarding the existing database like structure. The section is as follows.

```
Background: Initial plant catalog and purchase orders
  Given the following $Plants
    | _id | name           | description        | price  |
    |  1  | Mini excavator | Excavator 1.5 tons | 100.00 |
    |  2  | Mini excavator | Excavator 2.5 tons | 120.00 |
    |  3  | Midi excavator | Excavator 3.0 tons | 150.00 |
    |  4  | Maxi excavator | Excavator 5.0 tons | 200.00 |
  Given the following $PurchaseOrders
    | _id | plant          | startDate  | endDate    | cost    | poStatus |
    |  1  | {"_id": 1, …}¹ | 2016-02-29 | 2016-03-19 | 2000.00 | PENDING  |

   1. #{$toJson($Plants.findOne(1l))}
```

The name of the model is in each of the steps. We could iterate through the steps and obtain the corresponding models used in the application. The corresponding generated mock repository for PurchaseOrder model is given below.

```java
package com.example.models;

import org.springframework.data.repository.CrudRepository;

public interface MockPurchaseOrderRepository
      extends CrudRepository<PurchaseOrder, Long>{

}
```

Similarly, the repository for Plant also will be generated. This process makes Background section of Cucumber feature file mandatory for code generation. Even when there are no example rows, the steps should contain details about each of the models within the application. Once the mock repositories are generated, we can move forward to the mock controller generation.

The mock controller is used to verify the validity of the corresponding controller tests which are to be generated. We provide a static name for the mock controller as *Skeleton.java*. JSON Patch and JSON Merge Patch are being used in the approach. As a result, the corresponding methods are generated in a static way as they do not depend on the scenario or the application under test. An example of the generated code for JSON Merge Patch method is given below.

```java
public JsonNode $mergePatch(JsonNode obj, String json)
      throws Exception {
  JsonMergePatch mp = mapper.readValue(
      json, JsonMergePatch.class);
  return mp.apply(obj);
 }
```

Now we must setup the repositories within the mock in order to facilitate basic CRUD operations. We inject the repositories using the Spring Autowired annotation. Since we are generating the mock controller, we use the mock repositories instead of the actual JPA repositories. We make use of the Background section in Cucumber feature file to obtain the various model names and generate the corresponding repository declaration.

Also, it is required to implement the Background in features by generating code to save the examples provided. In order to facilitate this, using the generator, we convert each row under a step to a JSON format and create an array of JSONs for a single model. The corresponding generated code would look like shown below.

```java
@Autowired
MockPurchaseOrderRepository purchaseOrderRepo;

String[] purchaseOrderFixtures = {"{\"_id\":\"1\","
  + "\"plant\":#{$toJson($Plants.findOne(1l))},"
  + "\"startDate\":\"2016-02-29\",\"endDate\":\"2016-03-18\","
  + "\"cost\":\"2400.00\",\"poStatus\":\"PENDING\"}",
};
```

In the above JSON strings generated, we used the notations we had used in the Cucumber feature file like *$Plants*, *$toJson* etc. We need some mechanism in the mock to parse such notations. We use the Spring Expression Language (SpEL[17]) expression parser for this purpose. We define the database notations directly using an inner class. For example, by assigning *$PurchaseOrders* to the corresponding mock repository as shown below. We also define the $toJson method within the inner class and register the function.

```
class LocalSpelContext {
  public MockPurchaseOrderRepository $PurchaseOrders
      = purchaseOrderRepo;
  public String $toJson(Object o) throws Exception {
    return mapper.writeValueAsString(o);
  }
}

@RequestMapping("/initialize")
public void setupBackground() throws Exception {
  spelContext = new StandardEvaluationContext(
      new LocalSpelContext());
  spelParser = new SpelExpressionParser();
  spelContext.registerFunction("$toJson",
      LocalSpelContext.class.getDeclaredMethod(
            "$toJson", new Class[] { Object.class }));

  for (String json: purchaseOrderFixtures) {
      Expression expression = spelParser.parseExpression(
            json, new TemplateParserContext());
      String rewrittenJson = expression.getValue(
            spelContext, String.class);
      purchaseOrderRepo.save(mapper.readValue(
            rewrittenJson, PurchaseOrder.class));
  }
}
```

Once the initial setup is generated for the mock from the Background in the feature file, we concentrate on the Scenario Outlines which provides us each of the methods in the mock controller. Currently, we have created two Scenario Outlines. Let us first consider the Scenario Outline for "Creation of PurchaseOrder".

```
Scenario Outline: Creation of PurchaseOrder
  When customer calls 'createPO' using 'POST' on <uri> with <po>
  Then $PurchaseOrders must contain $mergePatch(<po>,<po_patch_merge>)
  And status code must be <status>
  And location must have <location>
  And 'poStatus' must be <poStatus>
  Examples:
  | uri  | po            | status | location        | po_patch_merge   | poStatus |
  | /pos | {"plant": …}1 | 201    | /pos/<po._id>   | {"id": …}2       | PENDING  |

   1. {"plant": #{$toJson($Plants.findOne(1l))}, "startDate": "2016-02-29", "endDate": "2016-03-04"}
   2. {"_id": #{$PurchaseOrders.count()+1}, "poStatus": "PENDING", "cost": "200"}}
```

[17] https://docs.spring.io/spring/docs/current/spring-framework-reference/html/expressions.html

From the title of the Scenario Outline, we get the model being used for the method. Hence in our case, it is PurchaseOrder. We could use a regular expression on the first step here to obtain the method name, the verb and the URI (URI value can be obtained from examples in the above case). We could also determine using the regular expression that if the step ends with “with <something>”, then the method has a RequestBody as a parameter. The name of the RequestBody parameter name is chosen as the name of the corresponding place-holder. Therefore, the initial method structure for createPO will be as shown below.

```
@RequestMapping(value="/pos",method=RequestMethod.POST)
public ResponseEntity<PurchaseOrder> createPO(
      @RequestBody String po) throws Exception{
}
```

Since we assume that after each execution of Scenario Outline, the database is reset, we add an extra parameter to check if the existing database should be kept alive for the method or not. Therefore, the method declaration structure changes slightly as follows.

```
@RequestMapping(value="/pos",method=RequestMethod.POST)
public ResponseEntity<PurchaseOrder> createPO(
      @RequestHeader("keepAlive") Boolean keepDbAlive,
      @RequestBody String po) throws Exception{
      if (!keepDbAlive) {
            setupBackground();
      }
}
```

As you can see that if `keepDbAlive` is false, we call the `setupBackground` method which resets the database with the initial database provided in the Background section of the feature.

Now we provide the following code when `RequestBody` parameter is not empty in order to parse the parameter into a JsonNode object.

```
JsonNode (<ReqBodyParam>1 = mapper.readTree(<ReqBodyParam>);
```

We JsonNode object should be updated with the Patch or Merge Patch expression and the resulting PurchaseOrder object should be derived. The corresponding code can be generated as follows.

```
Expression expression = spelParser.parseExpression(
      <po_patch or po_merge_patch>,
      new TemplateParserContext());
String expressionResult = expression.getValue(
      spelContext, String.class);
JsonNode mergePatchResult = <$mergePatch or $patch>(
      <JsonNode>, expressionResult);
PurchaseOrder _purchaseOrder = mapper.treeToValue(
      mergePatchResult, PurchaseOrder.class);
```

From the following step, we infer that we use *$mergePatch* and the expression used to apply on JsonNode is `po_merge_patch` value.

```
Then $PurchaseOrders must contain $mergePatch(<po>,<po_patch_merge>)
```

So, the above code, in this case, will be

```
Expression expression = spelParser.parseExpression(
      "{\"_id\": #{$PurchaseOrders.count()+1}, "
      + "\"poStatus\": \"PENDING\", \"cost\": \"200\"}}",
      new TemplateParserContext());
String expressionResult = expression.getValue(
      spelContext, String.class);
JsonNode mergePatchResult = $mergePatch(po1, expressionResult);
PurchaseOrder _purchaseOrder =
      mapper.treeToValue(mergePatchResult, PurchaseOrder.class);
```

Next to be done in this scenario is to generate code to save the derived PurchaseOrder object. Since it is a creation process, we provide a Link to the newly created object before saving. The corresponding location is added into headers and is returned using `ResponseEntity`. The status code provided the scenario is also returned.

```
purchaseOrder.add(new Link("<uri>/"
      + _purchaseOrder.get_id()));

purchaseOrderRepo.save(_purchaseOrder);

HttpHeaders headers = new HttpHeaders();
headers.add("Location", _purchaseOrder.getId().getHref());

return new ResponseEntity<PurchaseOrder>(_purchaseOrder,
      headers, HttpStatus.valueOf(<status>));
```

Now let us consider the second Scenario Outline “Processing of Pending PurchaseOrder”.

```
Scenario Outline: Processing of Pending PurchaseOrder
  When clerk calls <function_name> using <verb> on <uri>
  Then <po> should be '#{$PurchaseOrders.findOne(id)}'
  And $PurchaseOrders must contain $patch(<po>,<po_patch>)
  And status code must be <status>
  And 'poStatus' must be <poStatus>
  Examples:
  | function_name| verb  |uri               | status| po_patch    |id| poStatus|
  | acceptPO     | POST  |/pos/{id}/accept| 200   | [{"op": ...}]1 |1L| OPEN    |
  | rejectPO     | DELETE|/pos/{id}/accept| 200   | [{"op": ...}]2 |1L| REJECTED|

   1. [{"op": "replace", "path": "/poStatus", "value": "OPEN"}]
   2. [{"op": "replace", "path": "/poStatus", "value": "REJECTED"}]
```

As you can see, the first, third and fourth steps are identical to the previous Scenario Outline. But in this case, we do not have a RequestBody as there is no section like “with <something>” in the first step. Using regular expression, we parse the URI and check if it matches something inside curly brackets. If it matches, then the corresponding variable is provided as a path variable. In our case, we have the `id` as a path variable. Therefore, we get the initial structure of `acceptPO` method as follows.

```
@RequestMapping(value="/pos/{id}/accept",
      method=RequestMethod.POST)
public ResponseEntity<PurchaseOrder> acceptPO(
      @RequestHeader("keepAlive") Boolean keepDbAlive,
      @PathVariable Long id) throws Exception{
      if (!keepDbAlive) {
            setupBackground();
      }
}
```

Consider the case of the Scenario Outline “Creation of PurchaseOrder”. The steps for “Processing of Pending PurchaseOrder” are similar except that we have an additional step in this Scenario Outline.

```
Then <po> should be '#{$PurchaseOrders.findOne(id)}'
```

This is to obtain the existing PurchaseOrder object which should be accepted or rejected.

Correspondingly we generate a section of code to handle this. The generation is as follows.

```
Expression expression1 = null;
if (!keepDbAlive) {
  expression1 = spelParser.parseExpression(
      "#{$PurchaseOrders.findOne(1L)}",
      new TemplateParserContext());
} else {
  expression1 = spelParser.parseExpression(String.format(
      "#{$PurchaseOrders.findOne(new Long(%d))}", id),
      new TemplateParserContext());
}
PurchaseOrder _purchaseOrder1 = expression1.getValue(      spelCon-
text, PurchaseOrder.class);
JsonNode purchaseOrderJsonNode = mapper.valueToTree(
      _purchaseOrder1);
```

We get the expression and if keepDbAlive is false, then we use the id value from the examples. Else we replace it with the parameter value provided. Then it is passed into SpEL parser and we get the corresponding JSON string which is converted to PurchaseOrder object. This is further converted to JsonNode object in order to apply the patch. The acceptPO and rejectPO methods are almost similar. So I avoid to explain the generation of code for rejectPO. The complete generated mock controller covering the discussed scenario outlines is provided in Appendix D.

5.5 Generation of Test cases

Once the mock is generated, we would focus on the generation of test cases. As mentioned before, we would consider the Scenarios in the Cucumber feature file, which provides a sequence of Scenario Outlines for generating the test cases.

But before we consider the Scenarios, we can generate the static section of a spring test even without considering the feature file. The generated static structure of a test is as follows.

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(
      classes = DemoApplication.class)
@WebAppConfiguration
@DirtiesContext
public class SkeletonTest {
      @Autowired
      private WebApplicationContext wac;

      @Autowired
      @Qualifier("_halObjectMapper")
      ObjectMapper mapper;

      private MockMvc mockMvc;

      @Before
      public void setup() throws Exception {
            this.mockMvc = MockMvcBuilders.
                  webAppContextSetup(this.wac).build();
      }
}
```

Now let us consider the first Scenario.

```
Scenario: Create and accept PurchaseOrder
      When scenario "Creation of PurchaseOrder" with [1]
      And scenario "Processing of Pending PurchaseOrder" with [1]
```

The first step executes the Scenario Outline “Creation of PurchaseOrder”. The section “with [1]” can be ignored in this case as there is only a single example. This step requires a RequestBody object to be passed into the method. The value of this object is taken from the example value of placeholder in the Scenario Outline’s first step after “with” keyword. In this case, it is the `po`. The corresponding value can be obtained and assigned in the Test class. But to keep the Test class clean, we provide these values in a different Helper class. Therefore, the Helper class should be generated before generating the Test class.

The structure of Helper class is similar to the mock controller, till the definition of `setupBackground` method. Instead of the methods following `setupBackground`, we provide methods that return the object which is supposed to be the RequestBody used by the tests. In our case, we consider the Scenario Outline “Creation of PurchaseOrder”. The corresponding method to provide data will be generated as shown below.

```
public PurchaseOrder getCreatePOData() throws Exception{
  Expression expression = spelParser.parseExpression(
     "{\"plant\": #{$toJson($Plants.findOne(1l))}, \"startDate\":
     \"2016-02-29\", \"endDate\":
     \"2016-03-04\"}", new TemplateParserContext());
  String expressionResult = expression.getValue(spelContext,
     String.class);
  PurchaseOrder _purchaseOrder = mapper.readValue(
     expressionResult, PurchaseOrder.class);
  return _purchaseOrder;
}
```

The Helper should be injected in the Test class. Also, the `setupBackground` method should be called before the execution of each test case.

```
@Autowired
SkeletonHelper helper;

@Before
public void setup() throws Exception {
      this.mockMvc = MockMvcBuilders.
            webAppContextSetup(this.wac).build();
      helper.setupBackground();
      mockMvc.perform(post("/generated/initialize"));
}
```

Now, let us consider the scenario again. The scenario title will be used to create the test method name. We declare a MvcResult object in each test method. All the responses from the REST call are supposed to be received as MvcResult object. We also declare two objects of PurchaseOrder. One is to get the data of PO to be created from the helper class. The other is the created PO object. Therefore, the initial structure of the test method looks like follows.

```
@Test
public void testCreateAndAcceptPurchaseOrder() throws Exception {
      MvcResult result = null;
      PurchaseOrder _purchaseOrder = null;
      PurchaseOrder purchaseOrder = null;

      purchaseOrder = helper.getCreatePOData();
}
```

The URI for the corresponding method can be obtained from the corresponding Scenario Outline. In case the URI has some variable within curly brackets, we detect the presence of a path variable. If there is a path variable, then we assume that the path variable is the id of the object. The corresponding verb and status can also be obtained from the example in the Scenario Outline. We always take `keepDBAlive` value in the tests as true. This is because the database will be reset when we execute the second step of the scenario. Considering all these factors, the generated code to make the REST call corresponding to the first step will be as follows.

```
result = mockMvc.perform(post("/generated/pos")
            .header("keepAlive", true)
            .content(mapper.writeValueAsString(purchaseOrder))
            .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().is(201))
            .andReturn();
```

The resulting PurchaseOrder object will be assigned to `_purchaseOrder` and will be used for the next step in the scenario which is to accept the PO. In this case, we know that we have a path variable and there is no RequestBody. Correspondingly, the generated code for testing the acceptance of PO will be changed as shown below.

```
result = mockMvc.perform(post("/generated/pos/{id}/accept",
           purchaseOrder.get_id())
           .header("keepAlive", true))
           .andExpect(status().is(200))
           .andReturn();
```

The generated code for testing the rejection of PO will be as shown below.

```
result = mockMvc.perform(delete("/generated/pos/{id}/accept",
           purchaseOrder.get_id())
           .header("keepAlive", true))
           .andExpect(status().is(200))
           .andReturn();
```

The class containing the test cases covering the scenarios discussed in this section is provided in Appendix E.

Once we generate the mock and tests for the initial state diagram with the methods `createPO`, `acceptPO` and `rejectPO`, we would continue and implement the same approach to generate the mock controller methods and the corresponding tests for the complete state diagram. Therefore, the following additional methods will be generated in the mock controller.

- cancelPO
- closePO
- updatePO
- createPOExtension
- acceptPOExtension
- rejectPOExtension

Before the generation of these methods in the mock controller, it is required to make an additional entity `PurchaseOrderExtension` which contains the data to extend a Purchase Order namely the `endDate`. Else it will result in compilation errors after code generation.

The `updatePO` and `createPOExtension` methods would need request body objects for the tests. The corresponding methods to provide the request body will be generated in the Helper class.

We need to update the Scenarios in the Gherkin file to consider the above methods also into the tests. I gave the following scenarios in the final Gherkin file and the tests are generated for the corresponding scenarios.

- `Create accept and extend PurchaseOrder`
- `Create reject and cancel of PurchaseOrder`
- `Create accept and close of PurchaseOrder`
- `Create accept and cancel of PurchaseOrder`
- `Create reject and update of PurchaseOrder`
- `Create accept extend and reject extension of PurchaseOrder`

The mock controller, test class and the helper class provided in the Appendix do not consider the above scenarios. The complete structure of these classes can be obtained from the repository provided in Appendix F.

5.6 Discussion

In this chapter, we considered the scenario of equipment rental process in RentIT to implement the code generators. Using the state transitions of Purchase Order, we initially designed a Gherkin feature file, which was the chosen DSL. This Gherkin file was used to generate a mock controller using the low-level details, and then to generate a helper class, in order to provide the request body for the tests and finally to generate the test cases themselves.

In section 5.1, we implemented the implemented the Gherkin language for Purchase Order. In the following section, we described the grammar corresponding to a Gherkin language file. Before moving on to the code generation part, it was required to provide the domain model. The domain model was provided manually, so as to avoid making our DSL complicated by stuffing up details of the domain model in the Gherkin file. The mock controller generation is described in section 5.4. We also discussed the generation the mock as well as the actual repositories in this section. The following section dealt with the generation of test cases. The generation of a helper class was also discussed in this section which was used to provide the request body parameter values to the tests and thereby keeping our test class clean.

6 Case Study

The purpose of the thesis was to use a Model-Driven approach to generate test cases and the mock controller for a RESTful API. We evaluated the project by developing an actual controller using TDD approach by using the test cases generated. First, the initial state transitions which implemented `createPO`, `acceptPO` and `rejectPO` methods were verified. Once the verification was a success, the project was validated using the complete state transitions.

6.1 Test Evaluation

The initial scenarios covering creation, acceptance and rejection of Purchase Order are as follows.

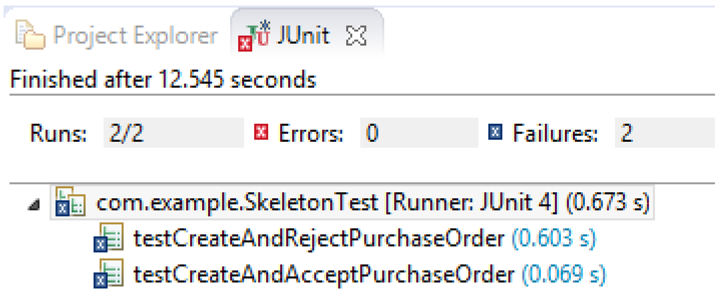
```
Scenario: Create and accept PurchaseOrder
      When scenario "Creation of PurchaseOrder" with [1]
      And scenario "Processing of Pending PurchaseOrder" with [1]

Scenario: Create and reject PurchaseOrder
      When scenario "Creation of PurchaseOrder" with [1]
      And scenario "Processing of Pending PurchaseOrder" with [2]
```

We provide the initial structure of the actual controller with the related repository declaration and run the test cases.

```
@RestController
@RequestMapping("/generated")
public class ActualController {
      @Autowired
      PlantRepository plantRepo;

      @Autowired
      PurchaseOrderRepository purchaseOrderRepo;
}
```



```
Finished after 12.545 seconds

Runs: 2/2    Errors: 0    Failures: 2

com.example.SkeletonTest [Runner: JUnit 4] (0.673 s)
    testCreateAndRejectPurchaseOrder (0.603 s)
    testCreateAndAcceptPurchaseOrder (0.069 s)
```

We can see that both the tests failed. As a result, we provide the structure of the methods.

```
@RequestMapping(value="/pos",method=RequestMethod.POST)
public ResponseEntity<PurchaseOrder> createPO(
      @RequestBody PurchaseOrder po) {
      HttpHeaders headers = new HttpHeaders();
      return new ResponseEntity<PurchaseOrder>(po, headers,
            HttpStatus.valueOf(201));
}
```

The tests are run again.

It throws an error at the assertion of `poStatus` value in the test. Therefore, we need to update the `poStatus` and save the `po`.

```
@RequestMapping(value="/pos",method=RequestMethod.POST)
public ResponseEntity<PurchaseOrder> createPO(
      @RequestBody PurchaseOrder po) {
    po.setPoStatus(POStatus.PENDING);
    po.calculateCost();
    po = purchaseOrderRepo.save(po);
    po.add(new Link("/pos/" + po.get_id()));

    HttpHeaders headers = new HttpHeaders();
    headers.add("Location", po.getId().getHref());

    return new ResponseEntity<PurchaseOrder>(po, headers,
      HttpStatus.valueOf(201));
}
```

On running the tests again, an assertion error occurs saying that the returned status is 405 (Method not allowed). We do not have a method to accept and reject Purchase Order. Therefore, we introduce these method structures with basic definition to the controller. The initial method structure for `acceptPO` is given below. The `rejectPO` method would have a similar structure.

```
@RequestMapping(value="/pos/{id}/accept",method=RequestMethod.POST)
public ResponseEntity<PurchaseOrder> acceptPO(
      @PathVariable Long id) throws Exception{
      PurchaseOrder po = purchaseOrderRepo.findOne(id);

      HttpHeaders headers = new HttpHeaders();

      return new ResponseEntity<PurchaseOrder>(po, headers,
            HttpStatus.valueOf(200));
}
```

This time on running the tests, we get assertion error regarding the updated `poStatus`. So the methods have to be updated with code to set the `poStatus` and the result must be saved to the database. So the resulting structure of `acceptPO` method is as follows.

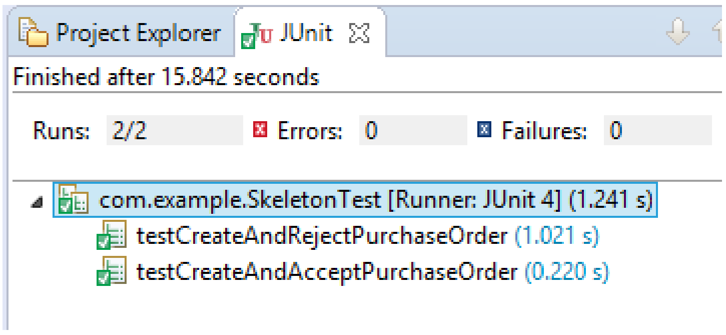
```
@RequestMapping(value="/pos/{id}/accept",method=RequestMethod.POST)
public ResponseEntity<PurchaseOrder> acceptPO(
      @PathVariable Long id) throws Exception{
      PurchaseOrder po = purchaseOrderRepo.findOne(id);
      po.setPoStatus(POStatus.OPEN);
      po = purchaseOrderRepo.save(po);

      HttpHeaders headers = new HttpHeaders();

      return new ResponseEntity<PurchaseOrder>(po, headers,
            HttpStatus.valueOf(200));
}
```

The structure of `rejectPO` will be similar to `acceptPO` method above.

Now on running the tests, we get the following result.



Finished after 15.842 seconds

Runs: 2/2 Errors: 0 Failures: 0

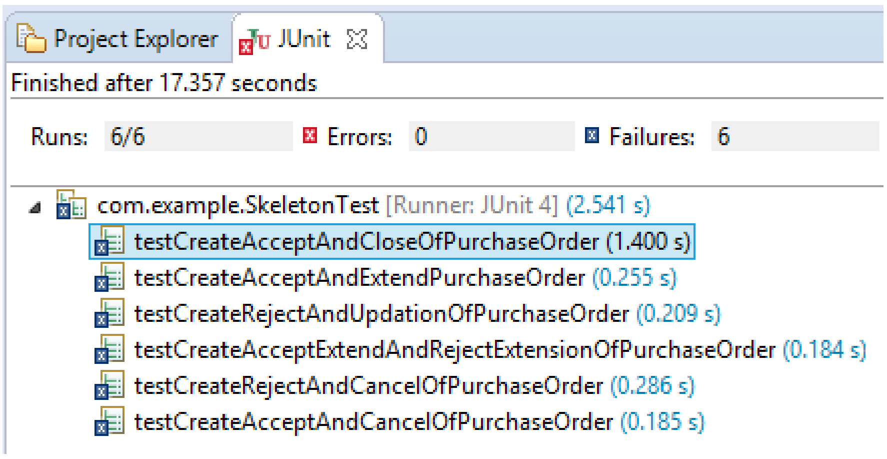
com.example.SkeletonTest [Runner: JUnit 4] (1.241 s)
testCreateAndRejectPurchaseOrder (1.021 s)
testCreateAndAcceptPurchaseOrder (0.220 s)

As you can see, all the tests have passed once we gave the complete implementation of `createPO`, `acceptPO` and `rejectPO` methods.

We have only verified the initial state transitions of Purchase Order. Now we will consider the complete application. We need to update the Gherkin feature file with the complete list of Purchase Order state transitions along with the following scenarios for testing.

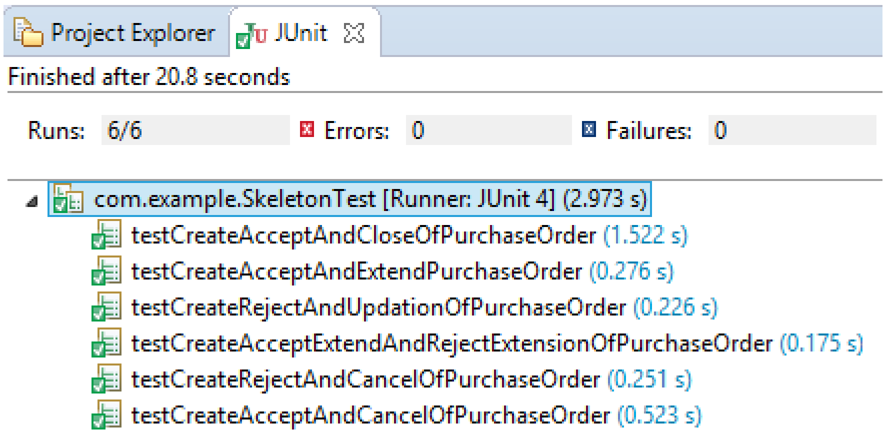
- `Create accept and extend PurchaseOrder`
- `Create reject and cancel of PurchaseOrder`
- `Create accept and close of PurchaseOrder`
- `Create accept and cancel of PurchaseOrder`
- `Create reject and update of PurchaseOrder`
- `Create accept extend and reject extension of PurchaseOrder`

On building the project, it would regenerate the mock controller, with additional mock methods based on the updated feature, the helper class, with additional methods to provide the test data, and finally the test class with the updated test cases provided by the scenarios in the feature. After the generating the updated test cases, we run them against the actual controller and get the following result in Eclipse.


```
Project Explorer   JUnit
Finished after 17.357 seconds
Runs: 6/6    Errors: 0    Failures: 6

com.example.SkeletonTest [Runner: JUnit 4] (2.541 s)
    testCreateAcceptAndCloseOfPurchaseOrder (1.400 s)
    testCreateAcceptAndExtendPurchaseOrder (0.255 s)
    testCreateRejectAndUpdationOfPurchaseOrder (0.209 s)
    testCreateAcceptExtendAndRejectExtensionOfPurchaseOrder (0.184 s)
    testCreateRejectAndCancelOfPurchaseOrder (0.286 s)
    testCreateAcceptAndCancelOfPurchaseOrder (0.185 s)
```

Again, just like the way we implemented before, we need to make the changes in the controller by defining the remaining methods. The method definition was done using the TDD way, by defining each method and verifying it with the tests. Finally, we got the following result.


```
Project Explorer   JUnit
Finished after 20.8 seconds
Runs: 6/6    Errors: 0    Failures: 0

com.example.SkeletonTest [Runner: JUnit 4] (2.973 s)
    testCreateAcceptAndCloseOfPurchaseOrder (1.522 s)
    testCreateAcceptAndExtendPurchaseOrder (0.276 s)
    testCreateRejectAndUpdationOfPurchaseOrder (0.226 s)
    testCreateAcceptExtendAndRejectExtensionOfPurchaseOrder (0.175 s)
    testCreateRejectAndCancelOfPurchaseOrder (0.251 s)
    testCreateAcceptAndCancelOfPurchaseOrder (0.523 s)
```

We can see the all the tests are passing. Thereby, we establish the validity of the tool and hence, the approach is considered successful.

6.2 Limitations

One of the main limitation to the approach is that the complete Cucumber feature has to be provided by the user in order to generate the mock controller and tests. The task can be tedious for larger REST applications. Therefore, we suggest the application of the tool mainly for simple REST APIs. Also, we have provided the flexibility to the user to provide the test data. As a result, the test data is also not generated but has to be provided in the Example section of each Scenario Outline.

The Background section, which is optional in a normal feature file, is a mandatory section for our tool. We used Background section to initialize the database before executing each Scenario Outline. But, it is also used for the tool to understand the various entities which are used in the application. As a result, even if the database for a particular entity is empty, a step has to be provided in the Background section for that entity with an empty database.

Another limitation is that the syntax for the step descriptions is quite strict. We used regular expressions to identify the type of step description. As a result, the descriptions should match with the existing feature file developed for Purchase Order.

7 Conclusion

In this thesis, an approach to automatically generate test cases as well as the mock controller using Model-Driven approach was discussed. For the proposed approach, Gherkin language was considered as the DSL so that it could be used to depict the resource interactions and the state transitions. Subsequently, a tool was developed to generate the test cases and the mock controller.

The mock controller generated has the most of the functionalities as the mocks generated by similar existing tools. Our mock controller can remain independent from the tests generated. It can be published just like any real controller and tested using script based testing tools like Postman. While the other tools like apiary mainly focus on executing one scenario at a time, which is also possible using our tool, we provide the functionality of chaining, which helps in executing multiple scenarios at a time by selecting one example from each scenario outline. Unlike other tools, which execute one method at a time and as a result, is completely disconnected from what the previous step did, the chaining functionality in our approach helps in connecting the method calls. Also, while testing the generated mock, it provides dynamic data compared to the other tools which responds with static mock data. For example, when we create an object through our mock controller, it returns the actual object which is saved in the database and not a static response object like what apiary does.

Similarly, the test generation tools for REST mainly tests a single scenario. They concentrate more on unit testing of each functionality. The same approach is implemented in our tool. Our tool is capable of performing unit testing considering a single scenario. Additionally, our tool is also capable of chaining tests to test multiple scenarios in a single test case. Thereby, we are expanding the testing process from simple unit testing to the much larger integration testing by facilitating the testing of longer sequences of calls. Moreover, in our approach, all the functionalities provided by the mock is tested by default by the generated test cases unless the information is not provided in the feature. As a result, we provide a way that will help the programmer to cover at least what is mocked.

Implementation of the DSL was done using Xtext and Xtend. Xtext provided a framework for defining the language grammar and Xtend facilitated the code generation. The generated test cases were first tested with the mock controller and the evaluation of the test cases was done by using these test cases to develop an actual application controller following the TDD approach.

7.1 Future Work

There is a lot of scope for improving the implementation and research for this approach in the future. First of all, there is a lot of room for optimizing the current approach and the implementation. During our approach, we did not concentrate on the performance of the tool, instead, we gave importance to the successful generation of tests and mock. As a result, a performance evaluation of the tool could be evaluated comparing with the other existing tools.

Currently, the created tool is integrated with Eclipse IDE. Eclipse is the most ideally used IDEs for Spring framework. Xtext is a well-integrated tool with Eclipse IDE. Since our tool was generated based on Xtext, it will run on all IDEs that can successfully integrate Xtext on them. Since Spring Tool Suite (STS) is Eclipse based, we assume that our tool would work on STS. Additional developments are going on regarding integrating Xtext with other

IDEs. The team behind Xtext have already managed to integrate Xtext with IntelliJ IDEA. So our tool may work even on IntelliJ and STS, but we leave that to be tested in the future.

The tool currently does not facilitate the generation of the various models and the actual controller. From the current Gherkin feature file, the attributes in a model can be obtained, but it does not provide information about their data types. A new DSL can be created for implementing the domain model. Xtext provides functionality to use multiple DSLs and thereby it could facilitate the generation of models. This can also be used to eliminate all the technical values used in the feature file like the method name, URI, verb etc. and make it completely human readable format. Such information can be provided either in the domain model DSL or a completely new DSL, and thereby, keeping the feature file clean. The challenge would come in the compliance of the domain model with the Gherkin feature. Even though this connection of the languages is possible in our approach, we did not focus on this and have left it for future research. Similarly, by providing some kind of Xtext parsable form of business logic for controller methods, the generation of the application controller can also be implemented. By adding these functionalities, the cost of development can be reduced as the developers need to concentrate mainly on writing a well-structured Cucumber feature file for the application.

Also, in the current approach, the specification file is ignored once the mock controller and the tests are generated. Research can be done by verifying the behaviour of the application by testing the feature file with the actual controller. The coverage that we have achieved through the controller can be presented it in the original feature by showing which section of the feature has been covered.

8 References

[1] R. T. Fielding, Architectural styles and the design of network-based software architectures, University of California, Irvine, 2000.

[2] T. Fertig and P. Braun, “Model-driven Testing of RESTful APIs,” *Proceedings of the 24th International Conference on World Wide Web,* pp. 1497-1502, 2015.

[3] R. Alarcon, E. Wilde and J. Bellido, “Hypermedia-driven RESTful service composition,” in *Service-Oriented Computing,* Springer Berlin Heidelberg, 2011, pp. 111-120.

[4] L. Richardson and S. Ruby, RESTful Web Services, O'Reilly Media, May 2007.

[5] “What's HTTP? Explain HTTP Request and HTTP Response,” 9 June 2008. [Online]. Available: http://geekexplains.blogspot.com.ee/2008/06/whats-http-explain-http-request-and.html. [Accessed 15 May 2016].

[6] P. Tahchiev, F. Leme, V. Massol and G. Gregory, JUnit in Action, Manning Publications, 2010, pp. 78-83.

[7] Z. Maamar, B. Benatallah and W. Mansoor, “Service Chart Diagrams - Description & Application,” in *Proceedings of the Alternate Tracks of The 12th International World Wide Web Conference,* May 2003.

[8] R. Swain, V. Panthi, P. K. Behera and D. P. Mohapatra, “Automatic Test case Generation From UML State Chart Diagram,” *International Journal of Computer Applications (0975 - 8887),* vol. 42, no. 7, pp. 26-36, March 2012.

[9] A. v. Deursen and P. Klint, “Domain-Specific Language Design Requires Feature Descriptions,” *Journal of Computing and Information Technology,* pp. 1-17, 2002.

[10] M. Wynne and A. Hellesoy, The Cucumber Book: Behaviour-Driven Development for Testers and Developers, The Pragmatic Bookshelf, January 2012.

[11] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold and P. McMinn, “An orchestrated survey of methodologies for automated software test case generation,” *Journal of Systems and Software,* vol. 86, no. 8, pp. 1978-2001, 2013.

[12] S. K. Chakrabarti and R. Rodriquez, “Connectedness Testing of RESTful Web-Services,” in *Proceedings of the 3rd India software engineering conference,* February 2010.

[13] U. Klein and K. S. Namjoshi, “Formalization and Automated Verification of RESTful Behavior,” in *Computer Aided Verification,* Springer Berlin Heidelberg, February 2011, pp. 541-556.

[14] S. K. Chakrabarti and P. Kumar, “Test-the-REST: An Approach to Testing RESTful Web-Services,” in *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD '09. Computation World:,* Athens, 2009, pp. 302-308.

[15] P. V. P. Pinheiro, A. T. Endo and A. Simao, “Model-Based Testing of RESTful Web Services Using UML Protocol State Machines,” in *Brazilian Workshop on Systematic and Automated Software Testing,* 2013.

[16] M. Dalgarno and M. Fowler, “UML vs. Domain-Specific Languages,” 2008. [Online]. Available: http://www.methodsandtools.com/archive/archive.php?id=71. [Accessed 15 May 2016].

[17] M. Dumas, “Order-to-Cash at RentIT,” [Online]. Available: https://courses.cs.ut.ee/MTAT.03.231/2016_spring/uploads/Main/RentIT-OrderToCash.pdf. [Accessed 15 May 2016].

[18] S. W. Suan, 16 December 2015. [Online]. Available: https://github.com/waisuan/SEED/blob/master/uom.ac.uk.msc.cucumber/src/uom/ac/uk/msc/cucumber/Gherkin.xtext. [Accessed 2016 May 15].

[19] C. Pautasso, O. Zimmermann and F. Leymann, “Restful web services vs. big'web services: making the right architectural decision,” *Proceedings of the 17th international conference on World Wide Web,* pp. 805-814, April 2008.

[20] M. Laitkorpi, P. Selonen and T. Systa, “Towards a model-driven process for designing restful web services,” in *Web Services, 2009. ICWS 2009. IEEE International Conference on,* Los Angeles, CA, 2009.

Appendix

I. Appendix A – RentIT Equipment Rental Process

The edited Purchase Order scenario for document discussion purpose which is adapted from [17].

RentIT is an equipment rental company (also known as a “plant hire” company) providing a wide range of construction equipment on demand, all the way from minor equipment items such as water pumps and drillers, to major equipment such as bulldozers, crawl dozers and cranes.

The process of renting equipments in RentIT starts when a new Purchase Order (PO) is received via its information system. A PO consists of a plant, and the corresponding start and end period of the rental. When a Purchase Order (PO) is received, a sales representative at RentIT checks the PO and the availability of the equipment requested in the PO. This may lead to one of two outcomes: (i) the PO is accepted; (ii) the PO is rejected, in which the customer is informed and the case is closed. In the latter case, the customer should update the PO and send a response within three days. If the customer does not respond within this delay, the PO is cancelled. A customer can send a request to cancel a PO, in which case the plant is freed up and the delivery is cancelled. A cancellation request must be received before the plant is dispatched from RentIT’s warehouse for delivery. Once the plant has been dispatched (i.e. it has left RenIT’s warehouse), it is no longer possible to accept the customer’s cancellation request.

Normally, the equipment is picked up on the end date indicated in the PO. It may happen however that the customer asks for an extension to the deadline by sending an updated purchase order (also known as a “PO extension”). When a PO extension asking for a deadline extension request is received, the sales rep checks if it is possible to grant the extension. If so, the extension request is accepted and the deadline extension is recorded in RentIT’s information system. If an extension is not possible, the request is rejected and the deadline remains unchanged. In both cases, the customer is informed.

II. Appendix B – Gherkin Feature file for PurchaseOrder

The Gherkin feature file for Purchase Order

The feature file contains only those scenarios discussed in the thesis report. The complete feature file covering all the Purchase Order state transitions can be obtained from the repository link provided in Appendix F.

```
Feature: PurchaseOrder feature
      As a customer
      In order to rent plant equipment
      I need to process a Purchase Order

Background: Initial plant catalog and purchase orders
  Given the following $Plants
    | _id | name           | description        | price  |
    |  1  | Mini excavator | Excavator 1.5 tons | 100.00 |
    |  2  | Mini excavator | Excavator 2.5 tons | 120.00 |
    |  3  | Midi excavator | Excavator 3.0 tons | 150.00 |
    |  4  | Maxi excavator | Excavator 5.0 tons | 200.00 |
  Given the following $PurchaseOrders
    | _id | plant           | startDate  | endDate    | cost    | poStatus |
    |  1  | {"_id": 1, …}[1] | 2016-02-29 | 2016-03-19 | 2000.00 | PENDING  |

Scenario Outline: Creation of PurchaseOrder
  When customer calls 'createPO' using 'POST' on <uri> with <po>
  Then $PurchaseOrders must contain $mergePatch(<po>,<po_patch_merge>)
  And status code must be <status>
  And location must have <location>
  And 'poStatus' must be <poStatus>
  Examples:
  | uri  | po              | status | location       | po_patch_merge  | poStatus |
  | /pos | {"plant": …}[2] | 201    | /pos/<po._id>  | {"id": …}[3]    | PENDING  |

Scenario Outline: Processing of Pending PurchaseOrder
  When clerk calls <function_name> using <verb> on <uri>
  Then <po> should be '#{$PurchaseOrders.findOne(id)}'
  And $PurchaseOrders must contain $patch(<po>,<po_patch>)
  And status code must be <status>
  And 'poStatus' must be <poStatus>
  Examples:
  | function_name| verb  |uri              | status| po_patch     |id| poStatus|
  | acceptPO     | POST  |/pos/{id}/accept| 200   | [{"op": …}][4] |1L| OPEN    |
  | rejectPO     | DELETE|/pos/{id}/accept| 200   | [{"op": …}][5] |1L| REJECTED|

Scenario: Create and accept PurchaseOrder
      When scenario "Creation of PurchaseOrder" with [1]
      And scenario "Processing of Pending PurchaseOrder" with [1]

Scenario: Create and reject PurchaseOrder
      When scenario "Creation of PurchaseOrder" with [1]
      And scenario "Processing of Pending PurchaseOrder" with [2]
```

1. #{$toJson($Plants.findOne(1l))}
2. {"plant": #{$toJson($Plants.findOne(1l))}, "startDate": "2016-02-29", "endDate": "2016-03-04"}
3. {"_id": #{$PurchaseOrders.count()+1}, "poStatus": "PENDING", "cost": "200"}}
4. [{"op": "replace", "path": "/poStatus", "value": "OPEN"}]
5. [{"op": "replace", "path": "/poStatus", "value": "REJECTED"}]

III. Appendix C – Grammar for a Gherkin Feature file

The Xtext grammar for a Gherkin feature file [18].

```
grammar org.xtext.example.feature.Feature with org.eclipse.xtext.common.Ter-
minals

generate feature "http://www.xtext.org/example/feature/Feature"

Feature:
      tags+=Tag*
      'Feature:'
      title=Title EOL+
      narrative=Narrative?
      background=Background?
      scenarios+=(Scenario | ScenarioOutline)+;

Background:
      'Background:'
      title=Title? EOL+
      narrative=Narrative?
      steps+=Step+;

Scenario:
      tags+=Tag*
      'Scenario:'
      title=Title EOL+
      narrative=Narrative?
      steps+=Step+;

ScenarioOutline:
      tags+=Tag*
      'Scenario Outline:'
      title=Title EOL+
      narrative=Narrative?
      steps+=Step+
      examples=Examples;

Step:
      stepKeyword=StepKeyword
      description=StepDescription EOL*
      tables+=Table*
      code=DocString?
      tables+=Table*;

Examples:
      'Examples:'
      title=Title? EOL+
      narrative=Narrative?
      table=Table;

Table:
      rows+=TABLE_ROW+ EOL*;

DocString:
      content=DOC_STRING EOL*;

enum StepKeyword:
      GIVEN = 'Given' | WHEN='When' | THEN='Then' | AND='And' | BUT='But'
;
```

```
Title:
      (WORD | NUMBER | STRING | PLACEHOLDER) (WORD | NUMBER | STRING | PLACE-
HOLDER  | TAGNAME)*;

Narrative:
      ((WORD | NUMBER | STRING | PLACEHOLDER) (WORD | NUMBER | STRING |
PLACEHOLDER  | TAGNAME)* EOL+)+;

StepDescription:
      (WORD | NUMBER | STRING | PLACEHOLDER  | TAGNAME)+;


BackgroundKeyword: 'Background:';

Tag: id=TAGNAME EOL?;

terminal NUMBER: '-'? ('0'..'9')+ ('.' ('0'..'9')+)?;

terminal PLACEHOLDER: '<' !('>' | ' ' | '\t' | '\n' | '\r')+ '>';

terminal TABLE_ROW: '|' (!('|' | '\n' | '\r')* '|')+ (' ' | '\t')* NL;

terminal DOC_STRING: ('"""' -> '"""' | "'''" -> "'''") NL;

terminal STRING:
      '"' ('\\' ('b' | 't' | 'n' | 'f' | 'r' | 'u' | '"' | "'" | '\\') |
!('\\' | '"' | '\r' | '\n'))* '"' |
      "'" ('\\' ('b' | 't' | 'n' | 'f' | 'r' | 'u' | '"' | "'" | '\\') |
!('\\' | "'" | '\r' | '\n'))* "'";

terminal SL_COMMENT: '#' !('\n' | '\r')* NL;

terminal TAGNAME: '@' !(' ' | '\t' | '\n' | '\r')+ ;

terminal WORD: !('@' | '|' | ' ' | '\t' | '\n' | '\r') !(' ' | '\t' | '\n' |
'\r')*;

terminal WS: (' ' | '\t');

terminal EOL: NL;

terminal fragment NL: ('\r'? '\n'?);
```

IV. Appendix D – Generated Mock Controller

The generated mock controller for Purchase Order

The following controller consider only the discussed scenarios within the thesis report. The complete mock controller covering all the state transitions of Purchase Order can be viewed from the repository link provided in Appendix F.

```
@Component
public class Skeleton {
  @Autowired
  MockPlantRepository plantRepo;

  @Autowired
  MockPurchaseOrderRepository purchaseOrderRepo;

  ObjectMapper mapper = new ObjectMapper().findAndRegisterModules();

  StandardEvaluationContext spelContext;
  ExpressionParser spelParser;

  class LocalSpelContext {
    public MockPlantRepository $Plants = plantRepo;
    public MockPurchaseOrderRepository $PurchaseOrders =
        purchaseOrderRepo;
    public String $toJson(Object o) throws Exception {
      return mapper.writeValueAsString(o);
    }
  }

  public JsonNode $mergePatch(JsonNode obj, String json)
      throws Exception {
    JsonMergePatch mp = mapper.readValue(json,JsonMergePatch.class);
    return mp.apply(obj);
  }

  public JsonNode $patch(JsonNode obj, String json) throws Exception {
    JsonPatch mp = mapper.readValue(json, JsonPatch.class);
    return mp.apply(obj);
  }

  String[] plantFixtures = {
    "{\"_id\":\"1\",\"name\":\"Mini excavator\",\"description\":"
    + "\"Excavator 1.5 tons\",\"price\":\"100.00\"}",
    "{\"_id\":\"2\",\"name\":\"Mini excavator\",\"description\":"
    + "\"Excavator 2.5 tons\",\"price\":\"120.00\"}",
    "{\"_id\":\"3\",\"name\":\"Midi excavator\",\"description\":"
    + "\"Excavator 3.0 tons\",\"price\":\"150.00\"}",
    "{\"_id\":\"4\",\"name\":\"Maxi excavator\",\"description\":"
    + "\"Excavator 5.0 tons\",\"price\":\"200.00\"}",
  };
  String[] purchaseOrderFixtures = {
    "{\"_id\":\"1\",\"plant\":#{$toJson($Plants.findOne(1l))},"
    + "\"startDate\":\"2016-02-29\",\"endDate\":\"2016-03-18\","
    + "\"cost\":\"2000.00\",\"poStatus\":\"PENDING\"}",
  };
```

```
@RequestMapping("/initialize")
public void setupBackground() throws Exception {
  spelContext = new StandardEvaluationContext(
      new LocalSpelContext());
  spelParser = new SpelExpressionParser();
  spelContext.registerFunction("$toJson",
      LocalSpelContext.class.getDeclaredMethod(
          "$toJson", new Class[] { Object.class }));

  for (String json: plantFixtures) {
    Expression expression = spelParser.parseExpression(
        json, new TemplateParserContext());
    String rewrittenJson = expression.getValue(
        spelContext, String.class);
    plantRepo.save(mapper.readValue(
        rewrittenJson, Plant.class));
  }

  for (String json: purchaseOrderFixtures) {
    Expression expression = spelParser.parseExpression(
        json, new TemplateParserContext());
    String rewrittenJson = expression.getValue(
        spelContext, String.class);
    purchaseOrderRepo.save(mapper.readValue(
        rewrittenJson, PurchaseOrder.class));
  }
}

@RequestMapping(value="/pos",method=RequestMethod.POST)
public ResponseEntity<PurchaseOrder> createPO(
    @RequestHeader("keepAlive") Boolean keepDbAlive,
    @RequestBody String po) throws Exception{
  if (!keepDbAlive) {
    setupBackground();
  }

  JsonNode po1 = mapper.readTree(po);

  Expression expression = spelParser.parseExpression(
      "{\"_id\": #{$PurchaseOrders.count()+1}, "
      + "\"poStatus\": \"PENDING\", \"cost\": \"200\"}}",
      new TemplateParserContext());
  String expressionResult = expression.getValue(
      spelContext, String.class);
  JsonNode mergePatchResult = $mergePatch(po1, expressionResult);
  PurchaseOrder _purchaseOrder = mapper.treeToValue(
      mergePatchResult, PurchaseOrder.class);
  _purchaseOrder.add(new Link("/pos/" + _purchaseOrder.get_id()));

  purchaseOrderRepo.save(_purchaseOrder);
```

```
    HttpHeaders headers = new HttpHeaders();
    headers.add("Location", _purchaseOrder.getId().getHref());

    return new ResponseEntity<PurchaseOrder>(_purchaseOrder,
        headers, HttpStatus.valueOf(201));
  }

  @RequestMapping(value="/pos/{id}/accept",method=RequestMethod.POST)
  public ResponseEntity<PurchaseOrder> acceptPO(
      @RequestHeader("keepAlive") Boolean keepDbAlive,
      @PathVariable Long id) throws Exception{
    if (!keepDbAlive) {
      setupBackground();
    }

    Expression expression1 = null;
    if (!keepDbAlive) {
      expression1 = spelParser.parseExpression(
          "#{$PurchaseOrders.findOne(1L)}",
          new TemplateParserContext());
    } else {
      expression1 = spelParser.parseExpression(String.format(
          "#{$PurchaseOrders.findOne(new Long(%d))}", id),
          new TemplateParserContext());
    }
    PurchaseOrder _purchaseOrder1 = expression1.getValue(
        spelContext, PurchaseOrder.class);
    JsonNode purchaseOrderJsonNode = mapper.valueToTree(
        _purchaseOrder1);

    Expression expression = spelParser.parseExpression(
        "[{\"op\": \"replace\", \"path\": \"/poStatus\", "
        + "\"value\": \"OPEN\"}]", new TemplateParserContext());
    String expressionResult = expression.getValue(
        spelContext, String.class);
    JsonNode mergePatchResult = $patch(purchaseOrderJsonNode,
        expressionResult);
    PurchaseOrder _purchaseOrder = mapper.treeToValue(
        mergePatchResult, PurchaseOrder.class);

    purchaseOrderRepo.save(_purchaseOrder);

    HttpHeaders headers = new HttpHeaders();
    return new ResponseEntity<PurchaseOrder>(_purchaseOrder,
        headers, HttpStatus.valueOf(200));
  }
  @RequestMapping(value="/pos/{id}/accept",
      method=RequestMethod.DELETE)
  public ResponseEntity<PurchaseOrder> rejectPO(
       @RequestHeader("keepAlive") Boolean keepDbAlive,
       @PathVariable Long id) throws Exception{
    if (!keepDbAlive) {
      setupBackground();
    }

    Expression expression1 = null;
    if (!keepDbAlive) {
      expression1 = spelParser.parseExpression(
          "#{$PurchaseOrders.findOne(1L)}",
          new TemplateParserContext());
    } else {
      expression1 = spelParser.parseExpression(String.format(
          "#{$PurchaseOrders.findOne(new Long(%d))}", id),
          new TemplateParserContext());
    }
    PurchaseOrder _purchaseOrder1 = expression1.getValue(
        spelContext, PurchaseOrder.class);
    JsonNode purchaseOrderJsonNode = mapper.valueToTree(
        _purchaseOrder1);


    Expression expression = spelParser.parseExpression(
        "[{\"op\": \"replace\", \"path\": "
        + "\"/poStatus\", \"value\": \"REJECTED\"}]",
        new TemplateParserContext());
    String expressionResult = expression.getValue(
        spelContext, String.class);
    JsonNode mergePatchResult = $patch(purchaseOrderJsonNode,
        expressionResult);
    PurchaseOrder _purchaseOrder = mapper.treeToValue(
        mergePatchResult, PurchaseOrder.class);

    purchaseOrderRepo.save(_purchaseOrder);

    HttpHeaders headers = new HttpHeaders();
    return new ResponseEntity<PurchaseOrder>(_purchaseOrder,
       headers, HttpStatus.valueOf(200));
  }
}
```

V. Appendix E – Generated Unit test cases

The generated test class for Purchase Order

The following test class consider only the discussed scenarios within the document. The complete test class with the complete test cases involving all the state transitions of Purchase Order can be viewed from the repository link provided in Appendix F.

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = DemoApplication.class)
@WebAppConfiguration
@DirtiesContext
public class SkeletonTest {
  @Autowired
  SkeletonHelper helper;

  @Autowired
  private WebApplicationContext wac;

  @Autowired
  @Qualifier("_halObjectMapper")
  ObjectMapper mapper;

  private MockMvc mockMvc;

  @Before
  public void setup() throws Exception {
    this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build();
    helper.setupBackground();
    mockMvc.perform(post("/generated/initialize"));
  }

  @Test
  public void testCreateAndAcceptPurchaseOrder() throws Exception {
    MvcResult result = null;
    PurchaseOrder _purchaseOrder = null;
    PurchaseOrder purchaseOrder = null;

    purchaseOrder = helper.getCreatePOData();
    result = mockMvc.perform(post("/generated/pos")
                .header("keepAlive", true)
                .content(mapper.writeValueAsString(purchaseOrder))
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().is(201))
                .andReturn();

    _purchaseOrder = mapper.readValue(result.getResponse()
                        .getContentAsString(), PurchaseOrder.class);
    Assert.assertThat(_purchaseOrder.getPoStatus().toString(),
      equalTo("PENDING"));

    result = mockMvc.perform(post("/generated/pos/{id}/accept",
                  _purchaseOrder.get_id())
                .header("keepAlive", true))
                .andExpect(status().is(200))
                .andReturn();

    _purchaseOrder = mapper.readValue(result.getResponse()
                        .getContentAsString(), PurchaseOrder.class);
    Assert.assertThat(_purchaseOrder.getPoStatus().toString(),
      equalTo("OPEN"));
  }
```

```java
  @Test
  public void testCreateAndRejectPurchaseOrder() throws Exception {
    MvcResult result = null;
    PurchaseOrder _purchaseOrder = null;
    PurchaseOrder purchaseOrder = null;

    purchaseOrder = helper.getCreatePOData();
    result = mockMvc.perform(post("/generated/pos")
                 .header("keepAlive", true)
                 .content(mapper.writeValueAsString(purchaseOrder))
                 .contentType(MediaType.APPLICATION_JSON))
                 .andExpect(status().is(201))
                 .andReturn();

    _purchaseOrder = mapper.readValue(result.getResponse()
                         .getContentAsString(), PurchaseOrder.class);
    Assert.assertThat(_purchaseOrder.getPoStatus().toString(),
      equalTo("PENDING"));

    result = mockMvc.perform(delete("/generated/pos/{id}/accept",
                   _purchaseOrder.get_id())
                 .header("keepAlive", true))
                 .andExpect(status().is(200))
                 .andReturn();

    _purchaseOrder = mapper.readValue(result.getResponse()
                         .getContentAsString(), PurchaseOrder.class);
    Assert.assertThat(_purchaseOrder.getPoStatus().toString(),
      equalTo("REJECTED"));
  }
}
```

VI. Appendix F – Prototype

The prototype code for the implemented tool along with a simple maven Java project where the generator is applied is submitted along with this thesis in an archived file. The source code is also maintained in a git repository which could be accessed at https://bitbucket.org/philipjohn007/restautomatictestgenerator.git. The source code can be cloned using the following command.

```
git clone https://philipjohn007@bitbucket.org/philipjohn007/restautomatictestgenerator.git
```

The repository contains two folders.

- Grammar and Generator
- DSL and Sample Generator Applied Project

The folder "Grammar and Generator" contains the Xtext project which contains the DSL grammar and the Xtend generator for generating code. The other folder contains a sample Maven Spring project which contains the Gherkin feature file (named as *PurchaseOrder.feature*) and the generated mock controller (Skeleton.java) and the generated test cases (SkeletonTest.java).

Running the generator code would require an Eclipse IDE configured with Xtext[18]. You need to run the generator as an Eclipse Application which would prompt the runtime environment of Eclipse IDE. Here you could create a project and provide the DSL in the form of a Gherkin feature file. For more information, you can refer to the tutorial screencast video using the URL https://youtu.be/ZeUUv5U0iJQ.

[18] https://eclipse.org/Xtext/download.html

VII. License

Non-exclusive licence to reproduce thesis and make thesis public