Journal of Applied Research and Technology

www.jart.icat.unam.mx

*Journal of Applied Research and Technology* 19 (2021) 676-693

Original

# REST API composition for effectively testing the Cloud

B. G. Wolde[a]* • A. S. Boltana[b]

[a]*Mekelle University, Ethiopian Institute of Technology-Mekelle (EiT-M), School of Computing, Mekelle, Ethiopia*
[b]*Director General for ICT Sector, Ministry of Innovation and Technology, Addis Ababa, Ethiopia*

Received 03 20 2020; accepted 01 06 2021
Available 12 31 2021

**Abstract:** The Cloud offers many ready-made REST services for the end users. This offer allows realizing a service level agreement (SLA) through the implementation of multiple services somewhere on the Internet. Hereby, ensuring SLA is important. Hence, software testing is a way of attesting quality assurance of a non-functional requirement from the end user's perspective. However, test engineering accesses only an interface that contains the high level behaviors without their underlying details. Testing such behaviors becomes an issue for classical testing procedures. This implies that REST API through composition be an alternative promising approach for modeling behaviors with parameters to test the Cloud. This effort aids to devise test effectiveness via REST-based behavior-driven implementation. It aims to understand functional behaviors through API methods based on input domain modeling (IDM). By making a careful REST design, the test engineer sends complete test inputs to its API directly on the application and gets responses from the infrastructure. This paper considers the NEMo mobility API specification to design an IDM, which represents pattern match of mobility search API path scope. In this way, sample mobility REST API service compositions are used to create test assertions for validating each path resource on a specified service.

*Keywords:* Testing the Cloud, mobility service composition, REST API, input domain modeling

*Corresponding author.
*E-mail address:* behailu.getachew@mu.edu.et (B. G. Wolde).

## 1. Introduction

Testing the Cloud is one of the most promising way to verify the quality of a cloud on the specified service (Gao et al., 2011). Through this way as a System Under Test (SUT), the Cloud provides an access to a test capability interface which enables a tester in ensuring an end-to-end functionality. This interface allows decoupling the behaviors of a system by abstracting from its underlying resources (for example., logic and back-end tiers). For instance, by abstracting the resources, the specified service is seamlessly free from having a responsibility to manage a state and provides a reusable interface regardless of language and platform. Under true context the Cloud follows a usage per pay model through accessing the required interface on a single virtual gateway via Internet technology (Blokland et al., 2013). Such accessing opportunity is letting us using the Cloud as a test platform environment. The primary motivation on this point is that an implemented cloud includes built-in actual artifacts like a usage-based cost, enhancing test execution, improving collaboration, behavior-driven interface (Blokland et al., 2013). Such artifacts base on pattern matching to offer the Cloud the capability that ensures the service level agreement (SLA), cost, enhancing test execution, improving collaboration, behavior-driven interface. Such artifacts base on pattern matching to offer the Cloud the capability that ensures the service level agreement (SLA), which encapsulates the non-functional aspects.

This offer helps to realize multiple low-level implementations through a high level matching interface within the test environment. With this interface it actualizes the access to a low-level privilege. *Usage-based* cost allows testing on demand-driven without extra charges or management with a high availability of resources. Enhancing test execution enables us to have a shorter testing time to complete test execution which reduces the efforts for test cost. *Improving collaboration* enhances the testing quality by distributing test units among different agile testing teams over the internet through an interactive *behavior-driven interface* (i.e., user stories). In this regard, use of the Cloud to support testing operations for assessing challenges associated with its software quality is another motivation. Challenges in software quality product is dynamic and vary in its nature of implementation for getting the access to service layers. This dynamism is relatively high from a cloud-computing perspective.

By definition, the challenges refer to an exposure of software or hardware component to the various failures or bugs. A bug can exist under two forms. One form is intended and a client-side error. Another form is often unintended and a server-side error. A bug that is intended is easily addressable whereas a bug that is not intended is hard and becomes a concern for quality assurance. The challenges are also sparsely located based on its nature from top to bottom layers in the Cloud. In this essence, the nature of these layers is described as an application (functionality), non-functional (or SLA) and infrastructure.

In (Blokland et al., 2013), a functionality challenge arises when the change happens on the application at the end user side. A non-functional challenge is usually known when the bug is exposed at least one good test case, and the infrastructure challenges become an issue when a problem primarily happens at the back-end components, e.g., a database or network infrastructure. Due to this distinct layering implementation, the failure presented in each layer demands its own way of testing for ensuring its quality assurance.

Quality assurance is a way of maintenance and prevention of a system from the bugs (Graham et al., 2008). Regardless of the reasons for the failure, for instance, a service that the Cloud offers often are a ready-made release from third-party software developers (Blokland et al., 2013). These ready-made software reduces the cost and time taking to create a new one. This is a great advantage for software developers. However, during revealing these software to be accessed by diversified clients they are not taking a proper testing time in software life cycle. This limited test time is one of the attributable factor for posing a challenge on the server-side, and it usually remains passive until it gets the favorable conditions to trigger it.

In such a case, the question is how to develop an effective promising approach for testing end-to-end SLA functionality? One way of guaranteeing this is to follow the rules and procedures of a black box implementation in terms of REST API service composition. A black box is a software testing strategy at the level of user requirements regardless of the source codes (Arcuri, 2019; Bertolino et al., 2010; Sneed & Verhoef, 2015). Thus, by this strategy exercising the data coverage through an interface is essential for testing the cloud service.

The rest of this paper is planned as below: next section discusses the motivations and its importance. Section 3 takes foundations which introduce the key terms definitions, REST service standard and API components. Section 4 elaborates the related approaches consisting of how to design an input domain model, REST API compositions (multiple API calls) to create a REST-based model-based test implementation. In section 5, the approach which creates a conceptual test system architecture that bases on a black box to search the matching pattern for REST service identifier. This identifier uses the contextual idea based on behavior-driven approach to execute NEMo sample instances like calling multiple testRouteFinder components by assuming different route ids generated from a Google map, and applying these routeids for testRouteConcatenator to compute the best route. Section 6 covers the NEMo project case study that is executed on a

NEMo mobility platform with REST specification model which aids to endow a behavior-driven development (BDD) based on domain specific test frameworks. The evaluation, in Section 7, stage includes a test object model diagram based on the conceptual test system architecture which enables to apply a sample test case specification, implementation, and execution. Following, in the same section, the results and discussions include the summary of findings. Finally, a conclusion explains the lessons learned and contributions with future work.

## 2. Motivations and their importance

In practice, the popular cloud providers offer the virtual gateway access to the distributed end users over this interface (Bertolino et al., 2010; Giessler et al., 2015). There are two service interface options to interact between cloud system and cloud users. The first option is a Representational State Transfer (REST) service, and the second option is a Simple Object Access Protocol (SOAP) service. The big providers like Amazon Web service, or Microsoft Azure cloud, use REST-like distributed architectural style and their documents prove this, that is, REST-based Application Program Interface (API). Commonly, this REST API becomes a useful approach to test cloud application using REST interface *(see Section 3 under definitions).* API is a specification that contains a set of standard testable functions with valid behaviors and parameters to be easily accessed and quickly established their data communication exchange. REST API testing is one of the most promising approach for testing a cloud through knowledge of model-driven software architecture.

In model-driven architecture, model-based testing (MBT) is a black box approach to design the test case generation based on the realization of functional behavior specification (Bertolino et al., 2010; da Silva, 2017). In the second option, the SOAP service is a standard protocol, but has overheads such as enveloping and messaging data at the time of exchanges. These overheads pose a high resource utilization that degrades the performance and gives less attention for using SOAP service than REST service by the providers for their service interface. In this paper, a NEMo mobility platform which runs REST services is used for execution. NEMo refers to "Sustainable satisfaction of mobility demands in rural regions" and offers the mobility services based on the requests of clients in that regions (Kuryazov et al., 2019). NEMo mobility services use a REST architectural style which consumes JavaScript Object Notation (JSON) open standard data interchange on the web (RFC 7159). One major task of this work is to build the appropriate approach through a practical experimental analysis on existing sample test cases taken from NEMo mobility services. As a test case writing environment, the client-side development with testing Framework is also considered. The objective is to understand the functional behaviors through API methods and validate them with executable behaviors of parameters based on input domain model using a specified mobility service.

## 3. Foundations

This section begins with key terms definitions like an API, REST, BDD, and DSL. In addition, it explains about the basic concepts that are the REST principles and API components.

### *3.1. Definitions*

The terms definitions given below are a brief descriptions in the context of this paper.

Definition 1: API is an application program interface that endows the capability to integrate a software system through composing the services (Angulo & Edwin, 2014; Ed-Douibi et al., 2018a). This API uses to connect the back-end system of the infrastructure via a cloud-native application development. The API capability also enables to provide compositions test effectiveness of the software system. The API capability also enables to provide compositions test effectiveness of the software system. An example in Figure 1 is indicated below with minimized API-based REST testing services like a RouteFinder(), RouteDetails(), and RouteConcatenator(). The diagram is a simple demonstration for visualization of REST API test model which will be further presented the in-depth elaborations of this in Sections 5 and 7.

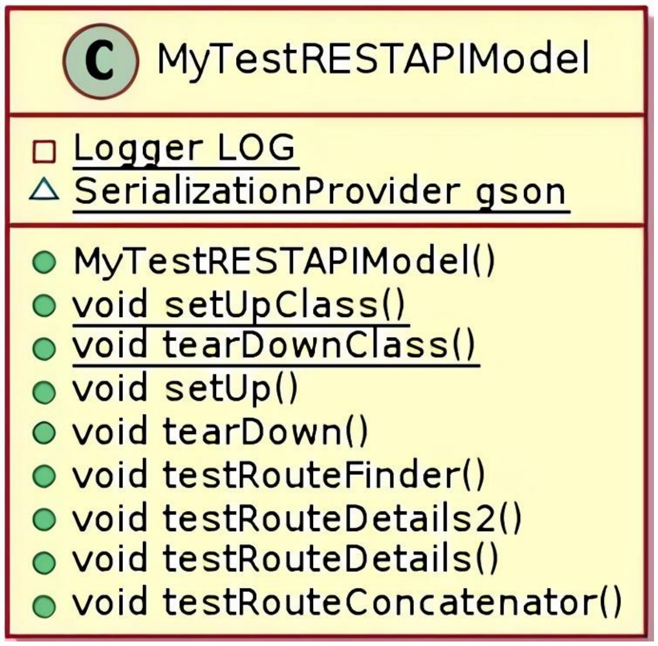


Figure 1. Example: Minimized API-based REST testing services (RouteFinder(), RouteDetails(), and RouteConcatenator()).

Definition 2: REST (REpresentational State Transfer) is an architectural style which means no official standard for web application that uses the RESTful web APIs (Fielding, 2000). A REST initializes with a null style like a World Wide Web (WWW). Every style extends from this null base concept like aJava class inherits from the super Object meta language. In Figure 1, the REST is a type of API that supports the implementation of the REST API compositions for executing such as two route ids (i.e., RouteDetails() and RouteDetails2()) via a route finder service to find route details services that are consumed by the route concatenate service. Meaning, multiple API calls are possible with API composer which transforms the discovered standard services into meta models for realization of language libraries for further code generation (Ed-Douibi et al., 2017; Izquierdo & Cabot, 2014).

Definition 3: Behavior-driven development (BDD) is an approach that supports a domain specific language (DSL) for testing a user acceptance using an agile development as a way to communicate among the stakeholders like a software developer, test engineer, business owners and customers (end users) (Behavior-drivendevelopment, 2020; Sivanandan, 2014) (see Table 1 in Section 5). Using this BDD principle a test engineer transforms each REST API into human-readable story. This user story helps to create a feature or test data like Table 1 in Section 5. This feature allows expressing scenarios with test steps in a Gherkin syntax (Given-When-Then) command. Each scenario is executed for validating semantics of each test steps using a cucumber test framework (Cucumber (software), 2020) which supports the BDD DSL approach in an Eclipse Integrated Development Environment (IDE) (Eclipse (software), 2020).

Definition 4: Domain Specific language (DSL) is an instance of meta model tool that aids to validate the models through transformation and synchronization like a Xtext plain textual language representation in (Wolde & Boltana, 2019) (see Figure 4 in Section 6). The Xtext is a language workbench consisting of structure that characterizes concepts (or classes), properties (attributes), operations (methods) and references (another concepts). The structure in this Xtext template is enabled by the workbench transformation engines to model, transform, validate and generate codes for test coverage execution in a model-driven architecture (MDA). The DSL structure has a pattern design that accepts inputs via attributes in an operation and receives the outputs from operations of the API components. From this, the multiple REST API calls can be connected concurrently without time and Geo-location constraints as the requests in demand (see Figure 3).

## *3.2. Basic Concepts*

The basic concepts cover the REST service standard with respect to the basic REST architectural principles including HTTP methods and API components.

### *3.2.1. REST Service Principles*

In (Fielding, 2000), REST works in a distributed environment through its capability of architectural style to use any protocol which is mainly the HTTP protocol for designing a loosely coupled application and a set of specifications to be used for developing web services. Architecturally, REST is a client-server in which a client side sends a data to a server then server side processes the data and returns responses. Both side communication builds the transfer state of resources. A resource is accessed through an endpoint, Universal Resource Identifier (URI). The design principles and guidelines for REST (Angulo & Edwin, 2014; Ed-Douibi et al., 2018b; Murphy et. al, 2017) are as follows: uniform interface, addressable, statelessness, layered system and code on demand.

Uniform Interface: REST enables the APIs interface for accessing a resource to the API consumers. A system that provides this resource has only one logical URI, and that specifies a way to get a related data. Addressable: REST data source works with the resources defined over URI. A standard interface has to be assigned to access the REST resources using the set of HTTP methods. Statelessness: Every state of transfer using REST is independent to the previous state which does not have overlap over the other one. Client session data is not dependent on the server side which maintains in that client requests only, and thereby, this architecture makes the REST design simple and lightweight. layered system: a REST-based service has multiple architectural layers. This layering supports an easy manipulation like addition, modification, and reorganizing through composition to meet the needs of evolve-ability on that service. Code On Demand: this capability allows some logic or behaviors to be isolated from a server to a client, to be run on the client. It enables customizing the web applications like a REST API generation through modeling the RESTful web service (Masse, 2011; Surwase, 2016).

The web application that describes the REST architecture is called a RESTful web service. RESTful web service uses the verbs to define HTTP methods. Once the address of the interface is specified, the API endpoints (that is, URI) is referred with these methods to retrieve ( i.e., GET), update (i.e., PUT), create (i.e., POST) and remove (i.e., DELETE) the resources. In computer programming, this API interface is a set of methods, protocols, and tools for implementing software applications. By using programming language, multiple libraries are implemented within frameworks to represent the actual behaviors.

### *3.2.2. API components*

In (Masse, 2011; Mulloy et al., 2013), the API components consist of libraries and frameworks. An API is a specification that describes the expected behavior while the library is the actual implementation of the business rules. One API can have more than one implementation with different libraries that share the same programming interface. A framework can rely on multiple libraries implemented the several APIs. This API favors the success to the behavior built into the framework which includes libraries to be used for extending its content with new plugin software tools into the framework itself. In the current definition, API components are usually web services that are designed and implemented inherently interoperability and are built with the explicit composed knowledge that they will interact with the distributed service consumers. In order to visualize and usable the REST API through web services, an explicit discovery of composed-based API is required. A typical example is the NEMo mobility services that compose a set of core services in an orchestrated manner like a *FindNearestStation, FindRoute, RouteDetails, RouteConcatenator,* and *RouteSelector* (Kuryazov et al., 2019).

These mobility operations provide a set of testable functions based on input domain modeling (Ammann & Offutt, 2016) for REST service composition. After an explicit interface is discovered and published for each composition, a hub or central coordinator acts as a managing one or more services to work together while the service consumers continue accessing each interface seamlessly independent of language and platform. The access to each resource depends on an input domain model to match with its API input pattern.

## 4. Related approaches

The related approaches include an API specification with input domain model and pattern matching diagram. In such a context, REST model-based testing is defined to aid the test case realization process through various approaches for generating executable test cases such as BDD approach with input model to create a test instances or DSLs. For this, two major relevant approaches are described below to design and derive an approach of testing the Cloud: with consecutive Arabic numbers within parenthesis.

• API specification: Input Domain model (IDM) and Pattern Matching, and

• REST-based model-based testing: test case realization process to the IDM.

Thus, this part helps to describe the combined techniques towards formulating the approach for validating the SUT (see Section 5).

### *4.1. API Specification: Input Domain Model (IDM) and Pattern Matching*

Input domain modeling (IDM) is a useful means to identify the testable functions and their parameters in (Ammann & Offutt, 2016; da Silva, 2017) which they are posed to affect the testing process. In (Ammann & Offutt, 2016; da Silva, 2017), IDM represents the range of input space of a SUT in an abstract way. A tester describes the signature of the input domain with the input properties. A tester builds a partition for each property. The partition is a set of blocks, each of which consists of a set of values. With the essence of a specific property, all values in each block are equivalent. A test input or test method is a tuple of values, one for the actual behaviors in each parameter. So, the test method uses exactly one block from each property. In essence, adding property with n blocks increases the number of combinations by a factor of n. So, in a practical scenario controlling the total number of combinations is a key task to input domain testing. After the IDM is created and values are assigned, a few combinations of values might be negative. The IDM must help the tester to identify and avoid or remove negative sub-combinations. With this, the IDM helps to design the properties based on realization of interface and functions. The former one depends on the parameters' realization. The latter one focuses on the actual behavior of the system rather than the interface. For test effectiveness, the preconditions and post-conditions are good sources of input properties. These conditions help to control the behaviors, which separate defined from undefined, within the chosen blocks and values. More blocks will have more tests, demanding more resources with high probable to find more bugs. Fewer blocks will have fewer tests, demanding lower resources with reduction in test effectiveness. For any given property one or two range of test input scopes are likely chosen to be applicable in (Ammann & Offutt, 2016):

• *Valid vs. invalid values*: Every partition must allow all (complete) values, whether valid or invalid.

• *Sub-block:* A range of valid values can usually be split ted into sub-blocks, such that each sub-block exercises a different part of operations.

• *Boundaries:* Values at or close to extremes often cause exceptions like a stress testing.

• *Happy path:* If the operational profile focuses heavily on middle values that are in between two extremes but not includes the boundary conditions.

• Enumerated types: A partition where blocks are a discrete, enumerated set often makes sense.

• Balance: From a cost perspective, it may be cheap or even free to add more blocks to the properties that have fewer blocks.

• Missing blocks: Check that the union of all blocks of a property completely covers the input space of that property.

• Overlapping blocks: Check that no value belongs to more than one block.

Assume an abstract partition p over some domain D. The partition p defines a set of equivalence classes (or blocks), $\boldsymbol{B_p}$. Together the complete block to avoid any missing element in D (Ammann & Offutt, 2016):

$$_p = \sum_{i=1}^{p} b_1, b_2, \ldots, b_p \tag{1}$$

$$b_i \cap b_j = 0, \forall_i \neq j, b_i, b_j \in B_p \cup b = D, b \in B_p \tag{2}$$

The blocks are pairwise disjoint, that is no element of D in more than one block:

$$b_{p_i} \cap b_{p_j} \cap b_{p_z} = 0, p_i \neq p_{ji} \neq p_z, \forall_i \neq j, b_i, b_j, b_{p_i}, b_{p_j}, b_{p_z} \in B_p \quad (3)$$

Using (1) and (2), a partition must fulfill the two formal properties:

•The p partition must cover the entire domain (completeness)

•The B blocks must not overlap (disjoint)

In Figure 2, Input D activity node starts an input message which has a fork point and creates three new activity nodes. The emerging new nodes form three blocks having each a set of values. Each block b for the partition p searches a matching pattern. A decision point for each b block in the p partition will check the preconditions to satisfy the searching. Once each block finds its own match then the matching will be computed through merging and join points to get the post condition. If the decision point becomes false to get its pattern matching then the activity transition becomes flow final node to exit the actions. With this respect, in Figure 1, the three p partitioning (that is, $p_i, p_j$ and $p_z$) bases class equivalence to build set of blocks or regions.

Class $p_i = p_{i-1}, p_{i-2}, p_{i-10} = 0,1,2,\ldots,9$.

The i th in p class takes only a digit value. Class $p_j$=a, b, ..., z and/or A, B, ..., Z. The jth in p class takes the alphabetic characters.

Class$p_z = \neg(A - Za - z0 - 9)$. The zth in p class takes the special characters. For example, in java language with class Pattern library the regular expression can be searched through passing it in a static compile method using a static object regex,i.e,Patternregex= Pattern.compile("[$\neg(A - Za - z0 - 9)$]");

Thus, an input model is needed to represent the functional behaviors in a modeling data such as API oriented REST compositions for testing the Cloud.

Each input key in the input model acts as an action to send a message to get its block size memory addressing, $b_p$ at i, j and z positions respectively. Then, each block like $b_{p_i}$, $b_{p_j}$, and $b_{p_z}$validates the truthiness for extracting the block matching pattern at each region, $b_{m_{i_i}}$, $b_{m_{j_j}}$and $b_{m_z}$. Before the final node in the activity model, the three searched blocks are merged and joined at one final activity node to return the post condition r=$b_{m_i} + b_{m_j} + b_{m_z}$for locating input pattern matching. In simlar fashion, once the searched pattern using the regex, this regex uses the Matcher object to find the pattern match of *str* parameterized values via the static matcher method, that is, Matcher ma=regex.matcher(str);

In the same way, in REST API computation this sequence of input action will be validated while the actual behaviors are executing from the libraries (implementation) that searches a pattern from the given API model to execute a REST data element such as a resource. Therefore, any test input violation to the given query for the required API implies a rule violation to the input model for accessing that resource, which is one of the motivational factors in this work.

The IDM approach uses the parameters as an interface and the behaviors as a function for making use of regular expressions for pattern matching. API design uses such pattern matching rules to describe a REST service composition. IDM also enables tester to examine the actual behaviors based on the rules set to represent the scenarios. Each behavior to its API in the scenarios has sequence of valid test steps which have the executable scripts for matching patterns in the given input (or testable) model. Through standard UML modeling language, the input model and the input pattern matching are described to show the input domain activity diagram as indicated in Figure 2. Based on this, each REST API in a mobility service has an input to be encoded from the standard keyboard, which constituents to make a set of block in each partition and a corresponding set of values. By designing the constituents, a valid combination for computing the matching pattern to its API method are implemented to execute based on the needs on the given URL path. This path will validate the given REST API which is related to the matching between the input values from the keyboard and the business values in the system to be tested. For the sake of demonstration, a sample HTTP method is used for testing NEMo mobility platform. This is done through implementation of test assertion in TestNG framework. This way, by using REST API testing, the behaviors of each test case are exercised, and the functionality of mobility services (REST API) are ensured with a model-based testing (MBT). RES-based MBT is the best option which allows creating the optimal range of test cases to its input domain model using black box testing approach.

## *4.2. REST-based model-based testing: Test case realization process*

By definition, model-based testing [MBT] follows a model-driven architecture for testing participants in practice with respect to the need of a model or a specification (Bertolino et al., 2010; Utting et al., 2012). The model simplifies the complexity by abstracting essential elements to its specification at the back-end of the Cloud. A cloud serves as a containment of implementation and execution over the infrastructure, which primarily includes REST application, web server, and back-end components. A model also wires these elements by using modeling language that has multiple modules as libraries implemented in the frameworks, which are deployed for the end users to have interactions and synchronization of data in the production environment. In model-based testing, test case realization process uses different approaches. With the favor of web technology, the end users prefer to use the published data via REST API.

Figure 3 represents the REST-based NEMo mobility web application and its realization approach for enabling the DSL instance from BDD meta-modeling. BDD creates the keywords derived from the requirements (see Table 1*)*. For this, REST API is a source to realize the various data model from simple JSON to complex JSON schema. In the realization process, the different tools are applied like a REST API composer, a postman as REST Client request and response, JSON2JSONSchema to get JSONSchema, BDD framework like a cucumber with Gherkin model to enable meta model for creating instance of DSL, web driver as test adaptation, and testing framework like a JUnit (de-facto for test component) and TestNG for test integration through executing the test implementation with visualization and interpretation. Accordingly, Figure 3 is authored and edited using a graphical tool with the Object Management Group (OMG) Business Process Modeling Notation (BPMN) 2.0 standard for designing a multiple REST API calls (Aagesen & Krogstie, 2015).

With this, the hollow uniheaded arrow refers to the data object as an input, which is a composed REST API. The round rectangle shape refers to various tool operations for the realization of various models such as data model, object model, BDD model, meta model, DSL and test adapter for test implementation. In each realization the corresponding output is expected. This output is presented with shaded uniheaded arrow business process. The upper long round rectangle highlights REST API service composition to data model realization where as the lower one shows the requirement to derive the BDD and IDM respectively for enabling the test realization process, and building test implementation and visualization.

In both ways, many authors have dealt with REST API composition approaches as described in (Asghari et al., 2018; Bellido et al., 2019; Ed-Douibi et al., 2018a; Ed-Douibi et al., 2017; Giessler et al., 2015; Neumann et al., 2018; Sangsanit et al., 2018). The findings in the papers are all at the conceptual levels that need an option to build a test system architecture especially for testing the Cloud, and a model-driven technical space to demonstrate test case specification with test implementation (see Table 2 and Figure 9). There are also works written in the testing services that do not have a relationship with issues for testing the Cloud rather they focus on the legacy web-based application in the desktop environment. In this context, the typical papers that support such testing (Arcuri, 2019; Bertolino et al., 2010; da Silva, 2017; Ed-Douibi et al., 2018b; Fertig & Braun, 2015; Ma et al., 2018; Sneed & Verhoef, 2015; Utting et al., 2012). In addition, all works associated with REST API do not consider the importance of IDM concepts and relationships for the target group to be tested as SUT. That means the research problem of addressing the technical spaces through model-based test architecture on the specified interface from the Cloud is not fully synchronized with a range of possible testing strategies. Thus, this paper raises an alternative and a new approach that supports both the software developer and customers on how to validate the REST product which is hosted in the cloud-based environment. Therefore, how does one realize given a test instances through BDD approach?

## 5. Approach

The approach depends on the requirements which meet the black box implementation to test the de-fined REST API service identifier (da Silva, 2017; Sivanandan, 2014). A black box focuses on behavior-driven descriptions of service features like a MBT to realize the domain specific language (DSL) as parameters and behaviors. Parameters include the attributes and types to specify an instance under a class like an object-oriented programming language. For example, in mobility service the *class Location* has a method *startLocation* with a parameter attribute *String* type. A test method or behavior (e.g., *Rout-eDetails*) and test object (e.g., *Route Plan*) are preconditions of this system model in the approach. The most promising agile solution to enable DSL is a BDD approach that captures the IDM based on model-based engineering (example, metamodeling using Gherkin syntax (Cucumber (software), 2020).

### *5.1. Gherkin model to perform domain specific language (DSL)*

Gherkin model is a way for enabling behaviors to realize the model through metamodeling concept, which creates an instance for domain specific language (DSL). By this DSL, a runner cucumber BDD framework using Eclipse plugin is scripted to simulate the Gherkin behaviors through REST API based on IDM (Cucumber (software), 2020; Smart, 2014). JUnit Framework is a defacto standard to test the components interacting with the behaviors executable in mobility service. In Figure 4, a JUnit test for component testing on Gherkin BDD is demonstrated. The test result validates the web driver browser Firefox together with cucumber test steps definitions implementation which is executed and displayed in the left preview console output of Eclipse IDE window. In the right preview Eclipse IDE window, the feature file shows the Gherkin model listed with keywords, which consist of scenarios and test steps Given-When-Then syntax, as is defined in Table 1.

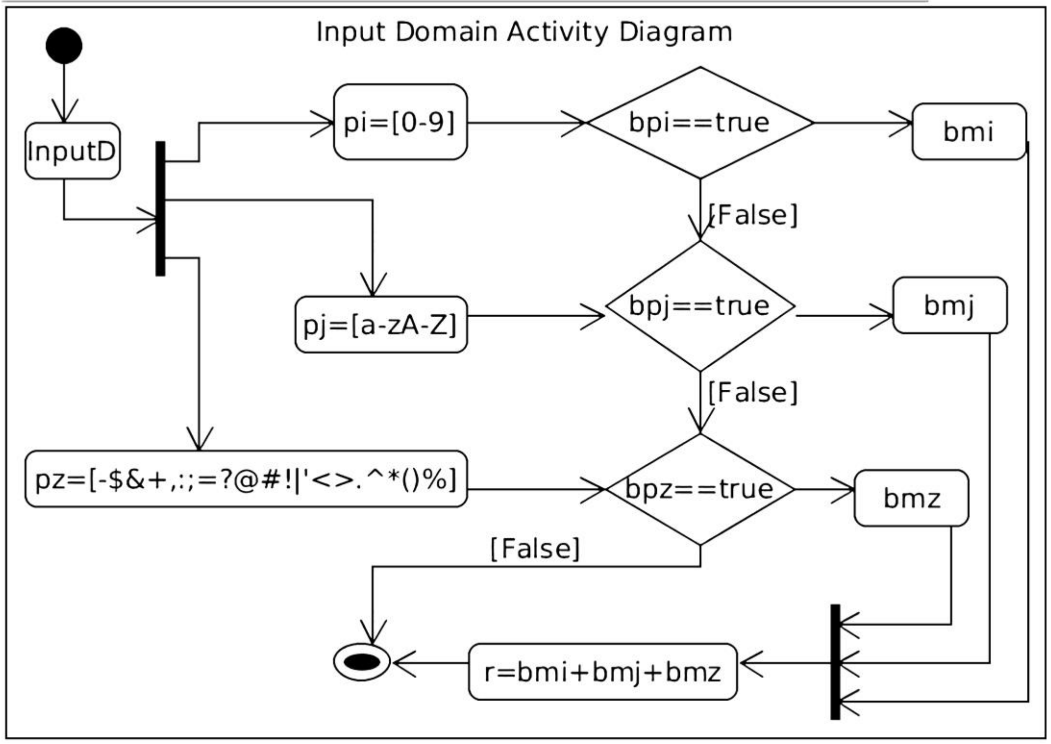


Figure 2. Input domain activity model for keyboard expression pattern matching.

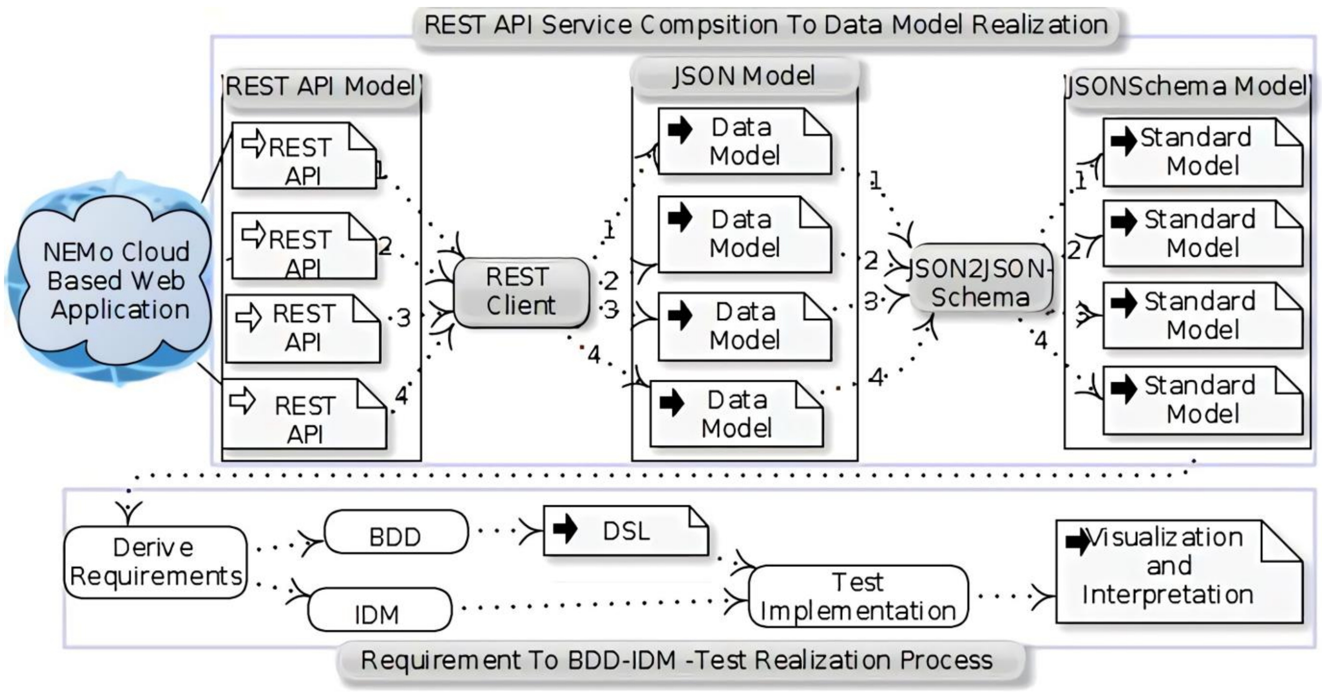
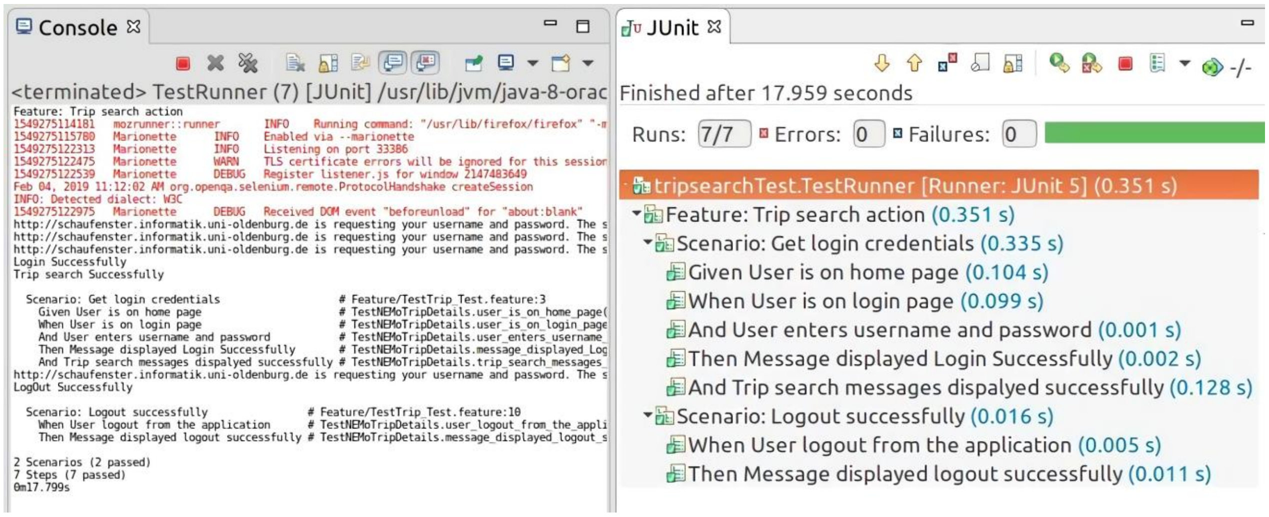


Figure 3. Multiple REST API calls using different stage realization approaches to derive BDD input domain test implementation.



```
<terminated> TestRunner (7) [JUnit] /usr/lib/jvm/java-8-orac
Feature: Trip search action
1549275114181   mozrunner::runner       INFO    Running command: "/usr/lib/firefox/firefox" "-m
1549275115780   Marionette      INFO    Enabled via --marionette
1549275122313   Marionette      INFO    Listening on port 33386
1549275122475   Marionette      WARN    TLS certificate errors will be ignored for this session
1549275122539   Marionette      DEBUG   Register listener.js for window 2147483649
Feb 04, 2019 11:12:02 AM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: W3C
1549275122975   Marionette      DEBUG   Received DOM event "beforeunload" for "about:blank"
http://schaufenster.informatik.uni-oldenburg.de is requesting your username and password. The s
http://schaufenster.informatik.uni-oldenburg.de is requesting your username and password. The s
http://schaufenster.informatik.uni-oldenburg.de is requesting your username and password. The s
Login Successfully
Trip search Successfully

  Scenario: Get login credentials                     # Feature/TestTrip_Test.feature:3
    Given User is on home page                        # TestNEMoTripDetails.user_is_on_home_page(
    When User is on login page                        # TestNEMoTripDetails.user_is_on_login_page
    And User enters username and password             # TestNEMoTripDetails.user_enters_username_
    Then Message displayed Login Successfully         # TestNEMoTripDetails.message_displayed_Log
    And Trip search messages dispalyed successfully # TestNEMoTripDetails.trip_search_messages_
http://schaufenster.informatik.uni-oldenburg.de is requesting your username and password. The s
LogOut Successfully

  Scenario: Logout successfully                   # Feature/TestTrip_Test.feature:10
    When User logout from the application         # TestNEMoTripDetails.user_logout_from_the_appli
    Then Message displayed logout successfully # TestNEMoTripDetails.message_displayed_logout_s

2 Scenarios (2 passed)
7 Steps (7 passed)
0m17.799s
```

```
Finished after 17.959 seconds
Runs: 7/7   Errors: 0   Failures: 0

tripsearchTest.TestRunner [Runner: JUnit 5] (0.351 s)
  Feature: Trip search action (0.351 s)
    Scenario: Get login credentials (0.335 s)
      Given User is on home page (0.104 s)
      When User is on login page (0.099 s)
      And User enters username and password (0.001 s)
      Then Message displayed Login Successfully (0.002 s)
      And Trip search messages dispalyed successfully (0.128 s)
    Scenario: Logout successfully (0.016 s)
      When User logout from the application (0.005 s)
      Then Message displayed logout successfully (0.011 s)
```

Figure 4. JUnit test assertion for DSL Gherkin model in cucumber framework using Eclipse IDE.

Table 1. Sample behavior-driven styles.

| Feature | Sample Test Data[1] | Test Steps |
|---|---|---|
| Trip Search in Action | **routeid:**33b9f7f6-efe5-429f-8bc8-32fb567b9eed<br>**start:** oe1e1fa7-0351-4e26-8d43-7134520481ea<br>**end:** b8a203d7-1ee3-4990-81af-391c77f97d42<br>**time:** 2018-09-16T12:00:24+02:00 | **Scenario:** Get login credentials<br><br>**Given** User is on home page.<br>**When** User is on login page.<br>**And** User enters username & password.<br>**Then** Message displayed login successfully.<br>**Scenario:** Logout successfully.<br><br>**When** User logout from the application.<br>**Then** Message displayed logout successfully. |

[1]The sample test data is taken from the NEMo API specification document, avaliable at Oldenburg University repository system.

### *5.2. Conceptual test system architecture*

Using this system architecture, a test model is described to support the validation of the test plan that is implemented in Figure 9. A textual and graphical symmetry visualization describe to represent how an API component initiates the required infrastructure in a given cloud system to specify the REST service composition for a Web API using a Hypertext Transfer Protocol (HTTP). The live preview visualization to switch from textual to graph form is done immediately as indicated with

In this case, the test system model-based architecture defines the major sections which are infrastructure, Composed Services, Route Composer, Composed Route Details, Route Concatenator,and Test Automation as defined in Figure 5.

• Infrastructure includes **DBType** as [DB] and **WebServer** as Web service to represent the mini-mized [cloud] component to describe NEMoCloudPlatform that contains the [API] for connecting the [NEMoMobility] composition REST [Services*] capable for accessibility and availability by the end users.

• Composed services are helpful to define the [Services*] components consisting of Service 1, Service 2, and Service 3, for instance. This opportunity allows enhancing the reliability through having an access to multiple APIs at the same time by the consumers of that service.

• Route composer uses the node [RouteFinder] component to find two or more services for enabling a capability mode (that is, *Bus, Walk* or *Bike*) based on the trip request operation to compute at least the three parameters like a time, start and end locations. After computing these parameters the composed route information is determined (see Figure 9).

• Composed route details mainly defines to include the **testRouteDetails** and **testRouteDe-tails2** to validate [RouteDetails] and [RouteDetails2] respectively. These composed route paths consist of sequences of composed stop stations for the capability modes which support the transportation or application in a NEMo mobility cloud system (see Figure 10).

• *Route Concatenator* is a component to support the **testRouteConcatenator** that consumes the Composed Route Details for validating each mode path based on the time capability to determine the best routes out of a given paths in the SUT. The inputs to this Route Concatenator are the outputs obtained from [RouteDetails2] and [RouteDetails]. Note that in the cloud system, tasks are executed in parallel, simultaneously, which are aided by the configuration of libraries in System Development kits (SDK) for Google map routes, places and directions. The details of these topics are not part of this paper.

• **Test automation** is the stage that enables using the **[TestModel]** to take the **[TestValidation]** action through executing a sample test instances from the **[CodeGeneration]** in a testing framework like a Test Next Generation (TestNG) (see Figure 9).

## 6. Application: NEMo project descriptions

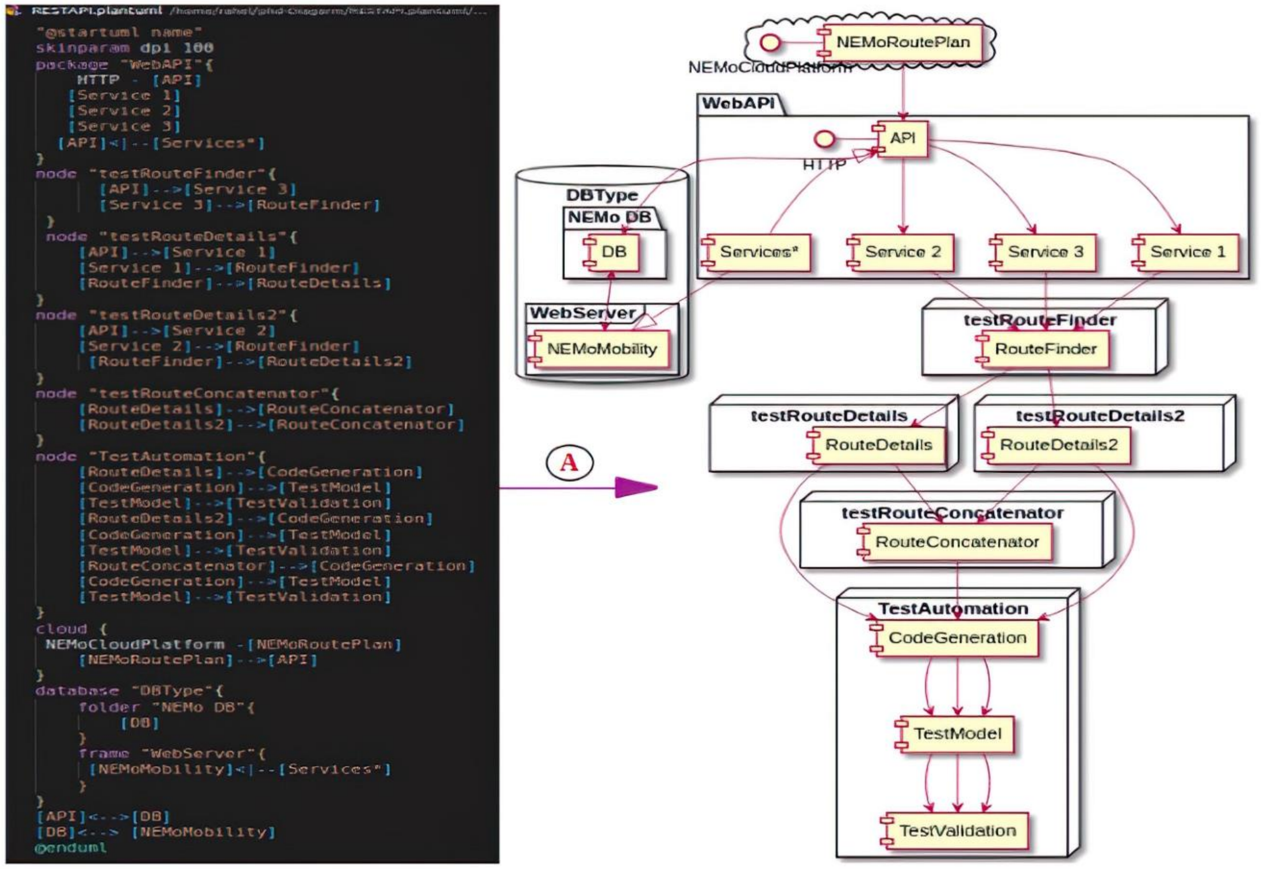
This section is about the application that is executed on NEMo Mobility Platform with respect to *NEMo essential libraries, NEMo In-Out sequence flow,* and *NEMo test instances.*

### *6.1. NEMo essential libraries*

NEMo project uses a JSON data representation seamlessly as communication between client and back-end via NEMo mobility platform as-a Service (PaaS). NEMo mobility platform gives interface libraries which use Java to implement services, mostly *IOData, RESTInvoker,* and *NEMoDatastructures* (Kuryazov et al., 2019). The IOData (i.e., InputOutput Data) specifies the input requested parameters such as coordinate points as locations and takes corresponding route id assigned from Google Direction identifier and then, passes them to the REST invoker framework to get JSONObject through ObjectMapper. This mapper class transforms to the specified JSONObject through JSON serialization process and stores the JSON data as a NEModata structure and returns as an output in the form of JSON data structure with key-value pairs at the client side through REST client service. This NEModata is also useful to create a data-driven testing since the In-Out sequence through action model transfers a state for each activity node till all the states terminate its In-Out sequence flows.

### *6.2. NEMo In-Out sequence flow*

In (Kuryazov et al., 2019), NEMo mobility platform primarily consists of four core REST mobility services to provide inter-modal (different capability mode) routing like a Bus, Bike and Walk which aim to achieve sustainable and flexible transportation mode based on demands to satisfy the customer. According to the Authors in 2019, NEMo action model is built for system mobility cloud platform with client inter-model traveler. In this model the In and Out activity to represent the input and output action-state sequence flow poses REST API service compositions through orchestrated data oriented model.


RESTAPI.plantuml /home/rahel/phd-Diagram/RESTAPI.plantuml/...

```
"@startuml name"
skinparam dpi 100
package "WebAPI"{
    HTTP - [API]
   [Service 1]
   [Service 2]
   [Service 3]
  [API]<|--[Services*]
}
node "testRouteFinder"{
        [API]-->[Service 3]
        [Service 3]-->[RouteFinder]
 }
 node "testRouteDetails"{
     [API]-->[Service 1]
     [Service 1]-->[RouteFinder]
     [RouteFinder]-->[RouteDetails]
}
node "testRouteDetails2"{
     [API]-->[Service 2]
     [Service 2]-->[RouteFinder]
      [RouteFinder]-->[RouteDetails2]
}
node "testRouteConcatenator"{
     [RouteDetails]-->[RouteConcatenator]
     [RouteDetails2]-->[RouteConcatenator]
}
node "TestAutomation"{
     [RouteDetails]-->[CodeGeneration]
     [CodeGeneration]-->[TestModel]
     [TestModel]-->[TestValidation]
     [RouteDetails2]-->[CodeGeneration]
     [CodeGeneration]-->[TestModel]
     [TestModel]-->[TestValidation]
     [RouteConcatenator]-->[CodeGeneration]
     [CodeGeneration]-->[TestModel]
     [TestModel]-->[TestValidation]
}
cloud {
 NEMoCloudPlatform -[NEMoRoutePlan]
     [NEMoRoutePlan]-->[API]
}
database "DBType"{
     folder "NEMo DB"{
         [DB]
     }
     frame "WebServer"{
      [NEMoMobility]<|--[Services*]
     }
}
[API]<-->[DB]
[DB]<--> [NEMoMobility]
@enduml
```

Figure 5. Test system model-based architecture.

Using this data model the standard schema model transformation is made. As it is designed in Figure 3 above, the transformed standard schema model from simple JSON model will be used to derive the user requirements. Each mobility data on the web via REST API is easily accessed during real time interaction between mobility cloud system and end users. To realize this interaction, the key NEMo core services are composed through service orchestration to enable the mobility inter-modal routing. In this regard, *FindNearestStation* is a NEMo service to find the nearest station given origin and destination states which are encoded to initiate the inter-modal routing. This routing enables to describe multiple modes concurrently. Based on this nearest station, *RouteFinder* NEMo service finds the routeid for accessing the *RouteDetails*. This routeid can have number of capability modes, which include many *sub routeids*. With this respect, *routeConcatenate* NEMo service evaluates the outputs of each *sub routeid*. At the end, *routeSelector* service considers the time parameter of each concatenated *routeid* to compute the best route (i.e., fastest) and return the result back to the client.

## *6.3. NEMo test instances*

Each NEMo mobility service is implemented with signatures that include service data points such as names, descriptions, input and output parameters, and its data types. These data fields are used to implement the NEMo services. Sample of two test instances are taken from the list of core mobility services for conducting evaluation in Section 7. Such instances are RouteFinder and RouteDetails. In a practical scenario, according to (Graham et al., 2008), an absence of fault does not mean that a software is free from bugs. That means under normal scenario an application may show success with a good test case that covers all IDM parameters interacting based on combinatorial testing approach (Jackson, 2017; Kuhn et al., 2016). However, this scenario outcome does not guarantee the quality to be sustained, and also an exhaustive testing is not feasible in practice (Graham et al., 2008). A sample of test input for each range of test data through input model is preferable. A good input model contains a data type with a range of values on the specified service.

On this point, a continuous testing is useful to consider the impact of negative threats on the service provided. In considering this issue, a predefined correct syntactic API specification that comprises high level descriptions of basic service elements is essential. This API should have an endpoint, input request, and output response. Endpoint is the entry point to access services based on its RESTinterface identifier. An identifier is an address of the service which contains baseURI plus an effective basePATH. baseURI refers to the server which serves as a common entry point whereas basePATH refers to a resource on the specified service.

The Scheme: HTTP://baseURI[:port]/basePath/, where the scheme is an HTTP, a port indicates a service (optional). Input request contains the test data and data types. For instance, the RouteFinder test request contains five parameters key-value pair composition as a test data to determine the subroute. The list of parameters are startLocation (string type), endLocation (string type), arrive (boolean type), time (date type), and transport Mode (string type). RouteDetails test request contains routeId as a test data which uses a String type. Output response refers to either expected or actual result. Thus, with the normal scenario, RouteFinder provides the expected JSON data as an output response which shows a route between origin and destination, and RouteDetails defines the JSON data as an output to show at least one complete route information (see Table 2 information).

## 7. Evaluation

The evaluation section mainly covers the efforts obtained during and after the approach formation followed by the results and discussions. Accordingly, first as a test setup, test case specification, execution, visualization and interpretation are explained. Test case specification contains the test plans which provide test requests (or test inputs), test data, expected and actual results. Once the test plan is arranged, the test implementation and adaptation are defined in a testing environment such as TestNG as testing automation, Eclipse IDE as development environment and third party libraries configurations. At the end a summary of results and discussions are made using concrete test object visualization diagram.

### *7.1. Test case specification*

In Table 2, the test case specification is adapted from the server side. This specification is part of the test plan to be used for implementation at the client side so that testing the cloud is done as intended. The test environment includes the cloud resource, which is run on a NEMo mobility platform. The client machine is used to implement the test driver based on the test case specification with behavior-driven styles in the behavior-driven development (BDD) framework. Based on client-side implementation, a tester sends a request through test script and receives a response from the web service, that is, NEMo mobility service.

### *7.2. Test model to create implementation*

This part uses two test cases (TC-1 and TC-2) for execution and visualization based on Table 2. It is conducted using TestNG testing framework with annotation @Test for both testRouteFinder() and testRouteDetails().

Table 2. Sample test case specification.

| TC# | Testable Functions | Sample Test Data | Expected | Actual |
|---|---|---|---|---|
| TC-1 | RouteFinder | **startLocation:** oe1e1fa7-0351-4e26-8d43-7134520481ea **endLocation:** b8a203d7-1ee3-4990-81af-391c77f97d42 **time:** 2018-09-16T12:00:24+02:00 **arrive:** true **mode:** bike | One full route[2] | |
| TC-2 | RouteDetails | **routeid:**33b9f7f6-efe5-429f-8bc8-32fb567b9eed or 05903d72-557f-4130-b009-dffe9055f33a | At least one best route[3] | |

### *7.2.1. Results in TC-1 and TC-2 discussions*

In Figure 6, the TC-1 test plan result is as expected to retrieve a real time route information. The test request from the client side is successful to send all five parameters as they are stated in Table 2 under second column. The actual result shows overflow exception, which is shaded in green color for **testRouteFinder** result. This is due to the logs exception on the web server as witnessed by the web administrator blogs and forums. Thus, with the TC-1 test plan, the test completed as normal scenario with PASSED status. The test functional

[2] Validated sug routes for each stop for evaluating the one full route start to end station.
[3] Time parameter is considered as capability during evaluating the best route.

completeness do not attain to cover all the required data accurately as expected, which misses to show a transport mode, BIKE. The finding helps to suggest that there is a need to maintain or update the web server at the level of server side. In Figure 7, the TC-2 test plan uses the routeid test data for testRouteDetails. It is a success in retrieving the expected response (one complete route from start to end location). The origin starts at stop Fuweg (label A in circle) station and the mobility starts, and then, wires with the coordinate points as latitude (lat) (53.14744) pair with longitude (lng) (8.19827) at label B, and continues to next stop till it reaches at the end locations coordinate point (53.14744-8.19827) at label C.

## 7.3. Object model for test implementation

Object modeling with sample test instances are important to precisely demonstrate the intended test system model architecture definition and usage. In line this *idea the model checking* is done for the two instance services such as RouteDetails and RouteDetails2 to compute a *RouteConcatenator* service. Accordingly, *MyTestRESTAPIModel* implements the test models for those test instances as indicated in Figure 8. This *MyTestRESTAPIModel* is stated by the numbers labeled (1, 2, 3, 4, 5, 6,and 7).

• 1-5 refer to represent the *MyTestRESTAPIModel* class on calling the API models for executing the *testRouteDetails* and *testRouteDetails2* via *testRouteFinder* which perform self validation first and follow to compute *testRouteConcatenator* (see Figure 9 and 10).

• 6 refers to show the ObjectMapper super class to generalize the Jackson Google GsonObjectMapper for enabling the realization of object serialization from JSON data model to Object Java Model.

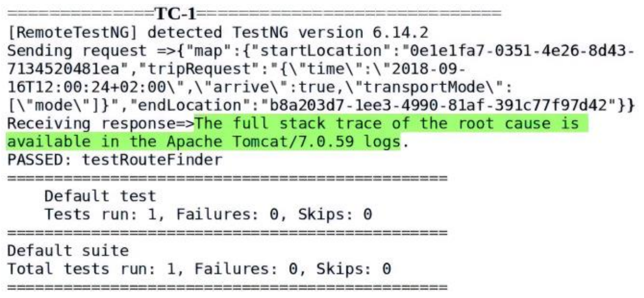
• 7 refers to the association between *GsonObjectMappe*r libraries and *MyTestRESTAPIModel* object to define and support the constructor method, MyTestRESTAPIModel(), to be instantiated for executions of the test instances (DSL models) in the validation stage.

## 7.4. Results and discussions

The results and discussions consider the execution and visualization which are followed the approach and test model implementation. For validity, the experimental analysis is made through test implementation of testRouteDetails and testRouteDetails2, and are executed to testtestRouteConcatenator using routeids (e.g., 05903d72-557f-4130-b009-dffe9055f33a and 33b9f7f6-efe5-429f-8bc8-32fb567b9eed) in the Mobility Route Plan System (see Figures 10 and 9).

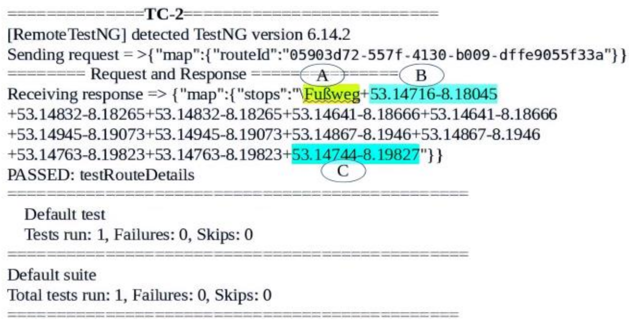
In Figure 9, the TestNG@Test annotation aims to validate mobility DSL REST libraries within the JSON API framework using concepts of BDD principles to define the behaviors like a Gherkin model Given-When-Then syntax in Table 1.

This test implementation in Figure 9 also evaluates indirectly the low level artifacts through high level descriptions in REST BDD framework. In Figure 10, all stops labeling such as stop1, stop2 and stop3 of capability mode would compose to get the best route during evaluation of testRouteConcatenator. For routeid: 05903d72-557f-4130- b009- dffe9055f33a, the best route is found at the stop station on location Fuweg, and for routeid: 33b9f7f6-efe5-429f-8bc8-32fb567b9eed, the best route is found at the stop station on location Radweg. In this regard, Figure 9 shows the partial view of TestNG Test Adaptation with @Test Annotation for RouteFinder and RouteDetails using REST API service composition routeid: 05903d72-557f-4130-b009- dffe9055f33a, the best route is found at the stop station on location Fuweg, and for routeid: 33b9f7f6-efe5-429f-8bc8-32fb567b9eed, the best route is found at the stop station on location Radweg. In this regard, Figure 9 shows the partial view of TestNG Test Adaptation with @Test Annotation for RouteFinder and RouteDetails usingREST API service composition.



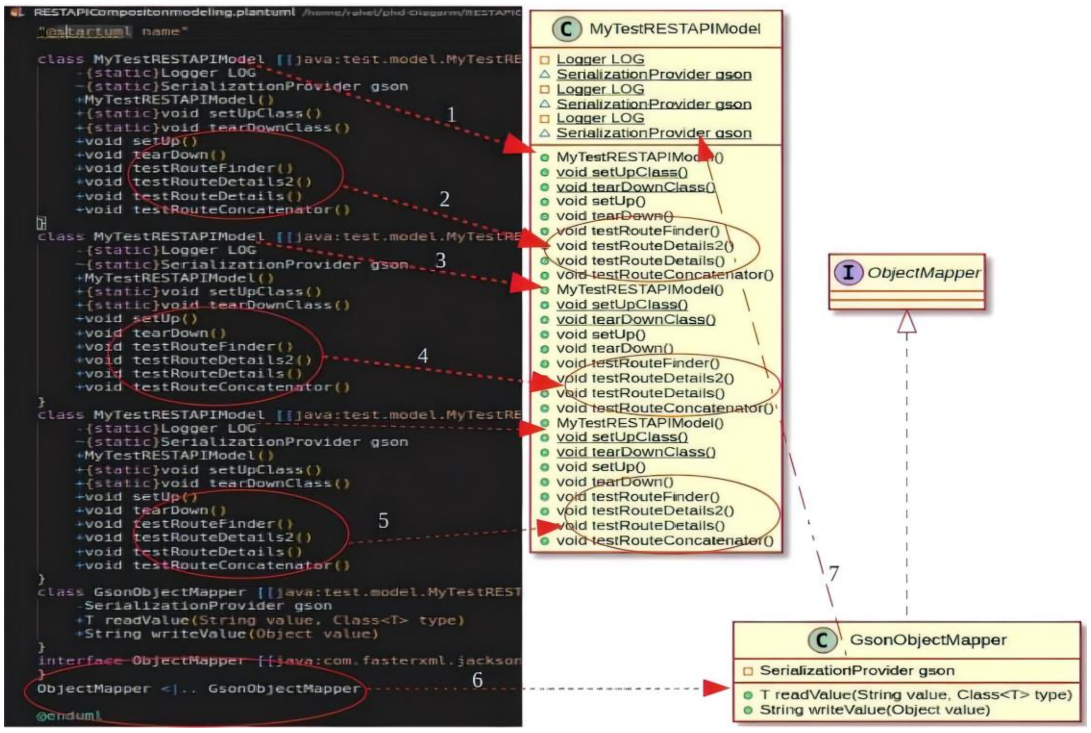
```
==============TC-1==============================
[RemoteTestNG] detected TestNG version 6.14.2
Sending request =>{"map":{"startLocation":"0e1e1fa7-0351-4e26-8d43-
7134520481ea","tripRequest":"{\"time\":\"2018-09-
16T12:00:24+02:00\",\"arrive\":true,\"transportMode\":
[\"mode\"]}","endLocation":"b8a203d7-1ee3-4990-81af-391c77f97d42"}}
Receiving response=>The full stack trace of the root cause is
available in the Apache Tomcat/7.0.59 logs.
PASSED: testRouteFinder
===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================
Default suite
Total tests run: 1, Failures: 0, Skips: 0
===============================================
```

Figure 6. TC-1 Test Result For RouteFinder end-to-end integration testing using TestNG framework.



```
==============TC-2===========================
[RemoteTestNG] detected TestNG version 6.14.2
Sending request = >{"map":{"routeId":"05903d72-557f-4130-b009-dffe9055f33a"}}
======== Request and Response =====A=======B
Receiving response => {"map":{"stops":"\Fußweg+53.14716-8.18045
+53.14832-8.18265+53.14832-8.18265+53.14641-8.18666+53.14641-8.18666
+53.14945-8.19073+53.14945-8.19073+53.14867-8.1946+53.14867-8.1946
+53.14763-8.19823+53.14763-8.19823+53.14744-8.19827"}}
PASSED: testRouteDetails                        C
===============================================
  Default test
  Tests run: 1, Failures: 0, Skips: 0
===============================================
Default suite
Total tests run: 1, Failures: 0, Skips: 0
===============================================
```

Figure 7. TC-2 Test Result For RouteDetails end-to-end Integration Testing using TestNG framework.

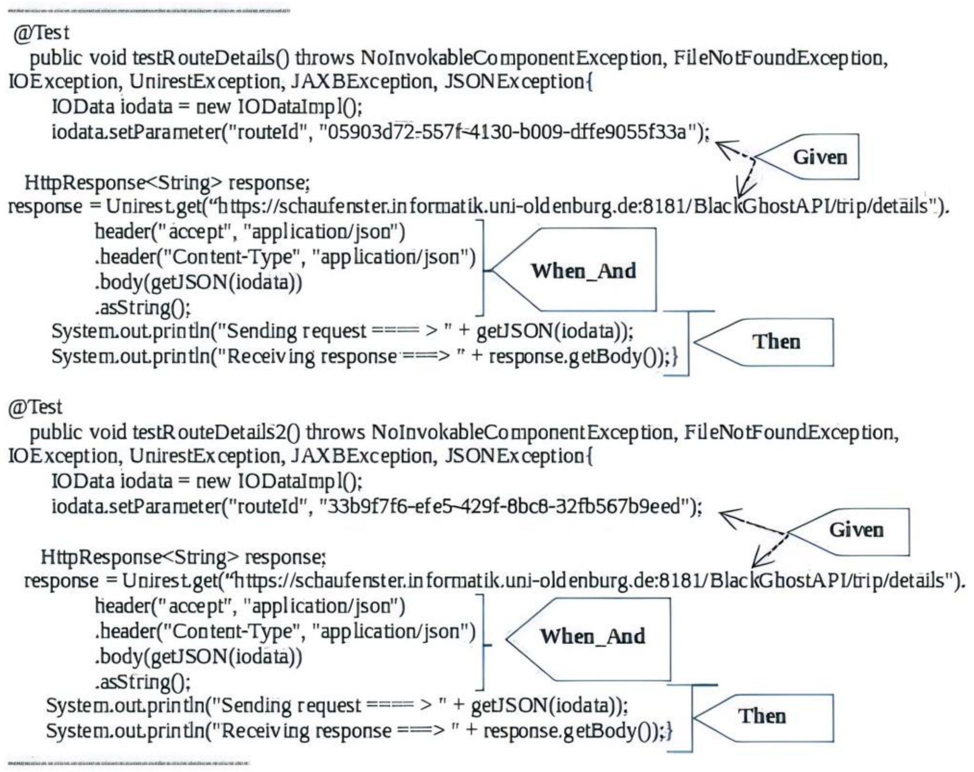

RESTAPICompositonmodeling.plantuml /home/rahel/phd-Diagarm/RESTAPIC
1
2
3
4
5
6
7
MyTestRESTAPIModel
Logger LOG
SerializationProvider gson
MyTestRESTAPIModel()
void setUpClass()
void tearDownClass()
void setUp()
void tearDown()
void testRouteFinder()
void testRouteDetails2()
void testRouteDetails()
void testRouteConcatenator()
ObjectMapper
GsonObjectMapper
SerializationProvider gson
T readValue(String value, Class<T> type)
String writeValue(Object value)

```
"@startuml name"

class MyTestRESTAPIModel [[java:test.model.MyTestRE
    -{static}Logger LOG
    -{static}SerializationProvider gson
    +MyTestRESTAPIModel()
    +{static}void setUpClass()
    +{static}void tearDownClass()
    +void setUp()
    +void tearDown()
    +void testRouteFinder()
    +void testRouteDetails2()
    +void testRouteDetails()
    +void testRouteConcatenator()
}
class MyTestRESTAPIModel [[java:test.model.MyTestRE
    -{static}Logger LOG
    -{static}SerializationProvider gson
    +MyTestRESTAPIModel()
    +{static}void setUpClass()
    +{static}void tearDownClass()
    +void setUp()
    +void tearDown()
    +void testRouteFinder()
    +void testRouteDetails2()
    +void testRouteDetails()
    +void testRouteConcatenator()
}
class MyTestRESTAPIModel [[java:test.model.MyTestRE
    -{static}Logger LOG
    -{static}SerializationProvider gson
    +MyTestRESTAPIModel()
    +{static}void setUpClass()
    +{static}void tearDownClass()
    +void setUp()
    +void tearDown()
    +void testRouteFinder()
    +void testRouteDetails2()
    +void testRouteDetails()
    +void testRouteConcatenator()
}
class GsonObjectMapper [[java:test.model.MyTestREST
    -SerializationProvider gson
    +T readValue(String value, Class<T> type)
    +String writeValue(Object value)
}
interface ObjectMapper [[java:com.fasterxml.jackson
}
ObjectMapper <|.. GsonObjectMapper

@enduml
```

Figure 8. Test assertion object modeling (model checking) implementation.


Given
When_And
Then
Given
When_And
Then

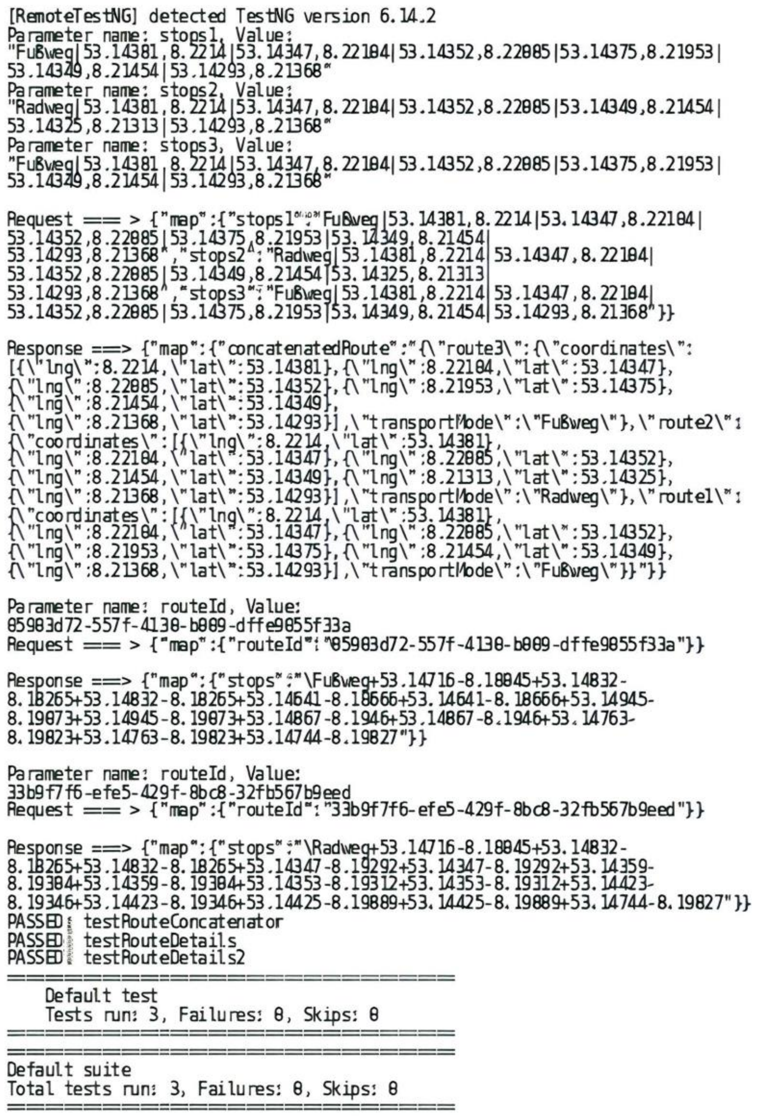
```
 @Test
   public void testRouteDetails() throws NoInvokableComponentException, FileNotFoundException,
IOException, UnirestException, JAXBException, JSONException{
       IOData iodata = new IODataImpl();
       iodata.setParameter("routeId", "05903d72-557f-4130-b009-dffe9055f33a");

   HttpResponse<String> response;
response = Unirest.get("https://schaufenster.informatik.uni-oldenburg.de:8181/BlackGhostAPI/trip/details").
            header("accept", "application/json")
            .header("Content-Type", "application/json")
            .body(getJSON(iodata))
            .asString();
       System.out.println("Sending request ==== > " + getJSON(iodata));
       System.out.println("Receiving response ===> " + response.getBody());}

@Test
   public void testRouteDetails2() throws NoInvokableComponentException, FileNotFoundException,
IOException, UnirestException, JAXBException, JSONException{
       IOData iodata = new IODataImpl();
       iodata.setParameter("routeId", "33b9f7f6-efe5-429f-8bc8-32fb567b9eed");

     HttpResponse<String> response;
  response = Unirest.get("https://schaufenster.informatik.uni-oldenburg.de:8181/BlackGhostAPI/trip/details").
            header("accept", "application/json")
            .header("Content-Type", "application/json")
            .body(getJSON(iodata))
            .asString();
      System.out.println("Sending request ==== > " + getJSON(iodata));
      System.out.println("Receiving response ===> " + response.getBody());}
```

Figure 9. TestNG test adaptation with @Test Annotation for RouteFinder and RouteDetails.



```
[RemoteTestNG] detected TestNG version 6.14.2
Parameter name: stops1, Value:
"Fußweg|53.14381,8.2214|53.14347,8.22104|53.14352,8.22085|53.14375,8.21953|
53.14349,8.21454|53.14293,8.21368"
Parameter name: stops2, Value:
"Radweg|53.14381,8.2214|53.14347,8.22104|53.14352,8.22085|53.14349,8.21454|
53.14325,8.21313|53.14293,8.21368"
Parameter name: stops3, Value:
"Fußweg|53.14381,8.2214|53.14347,8.22104|53.14352,8.22085|53.14375,8.21953|
53.14349,8.21454|53.14293,8.21368"

Request == > {"map":{"stops1":"Fußweg|53.14381,8.2214|53.14347,8.22104|
53.14352,8.22085|53.14375,8.21953|53.14349,8.21454|
53.14293,8.21368","stops2":"Radweg|53.14381,8.2214|53.14347,8.22104|
53.14352,8.22085|53.14349,8.21454|53.14325,8.21313|
53.14293,8.21368","stops3":"Fußweg|53.14381,8.2214|53.14347,8.22104|
53.14352,8.22085|53.14375,8.21953|53.14349,8.21454|53.14293,8.21368"}}

Response ==> {"map":{"concatenatedRoute":"{\"route3\":{\"coordinates\":
[{\"lng\":8.2214,\"lat\":53.14381},{\"lng\":8.22104,\"lat\":53.14347},
{\"lng\":8.22085,\"lat\":53.14352},{\"lng\":8.21953,\"lat\":53.14375},
{\"lng\":8.21454,\"lat\":53.14349},
{\"lng\":8.21368,\"lat\":53.14293}],\"transportMode\":\"Fußweg\"},\"route2\":
{\"coordinates\":[{\"lng\":8.2214,\"lat\":53.14381},
{\"lng\":8.22104,\"lat\":53.14347},{\"lng\":8.22085,\"lat\":53.14352},
{\"lng\":8.21454,\"lat\":53.14349},{\"lng\":8.21313,\"lat\":53.14325},
{\"lng\":8.21368,\"lat\":53.14293}],\"transportMode\":\"Radweg\"},\"route1\":
{\"coordinates\":[{\"lng\":8.2214,\"lat\":53.14381},
{\"lng\":8.22104,\"lat\":53.14347},{\"lng\":8.22085,\"lat\":53.14352},
{\"lng\":8.21953,\"lat\":53.14375},{\"lng\":8.21454,\"lat\":53.14349},
{\"lng\":8.21368,\"lat\":53.14293}],\"transportMode\":\"Fußweg\"}}"}}

Parameter name: routeId, Value:
05903d72-557f-4130-b009-dffe9055f33a
Request == > {"map":{"routeId":"05903d72-557f-4130-b009-dffe9055f33a"}}

Response ==> {"map":{"stops":"\Fußweg+53.14716-8.18045+53.14832-
8.18265+53.14832-8.18265+53.14641-8.18666+53.14641-8.18666+53.14945-
8.19073+53.14945-8.19073+53.14867-8.1946+53.14867-8.1946+53.14763-
8.19823+53.14763-8.19823+53.14744-8.19827"}}

Parameter name: routeId, Value:
33b9f7f6-efe5-429f-8bc8-32fb567b9eed
Request == > {"map":{"routeId":"33b9f7f6-efe5-429f-8bc8-32fb567b9eed"}}

Response ==> {"map":{"stops":"\Radweg+53.14716-8.18045+53.14832-
8.18265+53.14832-8.18265+53.14347-8.19292+53.14347-8.19292+53.14359-
8.19304+53.14359-8.19304+53.14353-8.19312+53.14353-8.19312+53.14423-
8.19346+53.14423-8.19346+53.14425-8.19889+53.14425-8.19889+53.14744-8.19827"}}
PASSED: testRouteConcatenator
PASSED: testRouteDetails
PASSED: testRouteDetails2
===============================================
    Default test
    Tests run: 3, Failures: 0, Skips: 0
===============================================
===============================================
Default suite
Total tests run: 3, Failures: 0, Skips: 0
===============================================
```

Figure 10. Test assertion (model checking) for RouteConcatenator

## 8. Conclusions

The summary of this paper includes the lessons learned and its contributions.

### *8.1. Lessons learned*

In this paper, the lessons basically introduce about the challenges and the promising approaches related to testing the Cloud and give its motivational importance through comparing against legacy testing procedure. The promising approach to testing the Cloud in REST API is the model-based black box testing strategy through REST API service composition running on NEMo mobility platform. In REST API testing with respect to the access to the behaviors implemented with libraries within API framework. This work also underlines the importance of input-output data coverage to derive test data for input request and output response based on IDM via REST API.

On this point, the evaluation is conducted using the three test instances of mobility services (i.e., testRouteFinder(), testRouteDetails() and testRouteConcatenator()). The test results show that the REST API has the capability to help the testers to be involved directly on the application by making test inputs to its API, and also getting an effective test response from the infrastructure via application domain of the web server. This response is represented through REST API Modeling Language (RAML) that supports the JSON data exchange from web server like Apache Tomcat running on GlassFish Application server v5. The findings elaborates a promising result with a few test plans for demonstrating the test effectiveness. The visualization from test implementation through execution of test runner agent in the testing frameworks are also imperative for motivating towards an extension of this work.

### *8.2. Contributions*

Its main contribution is to design the most promising approach with REST API model-based testing in the context of behavior-driven styles based on a black box strategy and show an end-to-end testing of the cloud-based system through targeting an effective REST API test input service composition. Since this research work is on the basis of REST service principle that includes API specification as defined in OMG. By using this standard principle several libraries implementation into API frameworks enable the access to be the executable behaviors isolated from the NEMo mobility services. The offer for selecting the best route is made through Google Cloud map integrated with enterprise cloud, which is Web Service Oxygen (WSO2). In addition, this work is able to show how to define an effective testing approach through REST service composition in MBT for testing the Cloud.

Furthermore, the study shows the applicability of this approach that testers have the chance to ap-ply a complete coverage with reasonable input model scopes or test case specification derived to test remotely the Cloud. This way, the software engineers will get a chance to improve the product by making detailed quality assurance and coverage analysis reporting. A run-time visualization technique to the client side at the end user is also attained through a simulation real time events at the server side in the Cloud. In general, this paper addresses four major aspects of contributions as below:

- Input Action Pattern Matching Analysis (See Figure 2).
- Designing REST API Composition Input Model (See Figure 3).
- Enabling Conceptual Test System Architecture (see Figure 5).
- Modeling and Implementing Concrete Test Automation Model (see Figure 8)

Therefore, the contributions of this work are multiples and useful for the scientific world especially for professionals as well as for business analysts. As a future work, this REST API testing covers the several aspects to be extended for covering all composed core NEMo mobility services with the range of optimal test data supplemented with appropriate artificial intelligence algorithms. Detailed test assertions should also be considered along with model checking to evaluate a cloud-based system like a NEMo cloud mobility platform as System Under Test (SUT).

## Acknowledgments

We are highly indebted and credited by the gracious help from University of Oldenburg, Software Engineering Group for their constant support for enabling this paper completion. We would also like to thank for the DAAD scholarship Grant for Research together with the Ethiopian-Engineering Capacity Building Program (EECBP). We are also thankful to the reviewers of this paper to reach this fruitful end. The research leading to these results has received funding from EECBP grant 57251549.

## References

Aagesen, G., & Krogstie, J. (2014). BPMN 2.0 for Modeling Business Processes. *Handbook on Business Process Management 1*, 219–250.
https://doi.org/10.1007/978-3-642-45100-3_10

Ammann, P., & Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press.
https://doi.org/10.1017/9781316771273

Angulo, B., & Edwin, J. (2014). *Dynamic composition of rest services.* [Thesis (Doctor in Engineering Sciences)-Pontificia Universidad Católica de Chile]

Arcuri, A. (2019). RESTful API automated test case generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM), 28*(1), 1-37.
https://doi.org/10.1145/3293455

Asghari, P., Rahmani, A. M., & Javadi, H. H. S. (2018). Service composition approaches in IoT: A systematic review. *Journal of Network and Computer Applications, 120*, 61-77.
https://doi.org/10.1016/j.jnca.2018.07.013

Behavior-drivendevelopment. (2020, September 14). In Wikipedia, The Free Encyclopedia. Retrieved: September 14, 2020, from:
https://en.wikipedia.org/wiki/behaviordrivendevelopment

Bellido, J., Alarcon, R., Pautasso, C., & Vairetti, C. (2019). SAW-Q: a dynamic composition approach of REST services based on queue model. *International Journal of Web and Grid Services, 15*(1), 29-58.
https://doi.org/10.1504/IJWGS.2019.096555

Bertolino, A., Grieskamp, W., Hierons, R., Le Traon, Y., Legeard, B., Muccini, H., ... & Tretmans, J. (2010). Model-based testing for the cloud. In Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Blokland, K., Mengerink, J., & Pol, M. (2013). *Testing cloud services: how to test SaaS, PaaS & IaaS*. Rocky Nook, Inc.

Cucumber, (software) (2020).
https://en.wikipedia.org/wiki/Cucumber_(software)

da Silva, R. V. E. (2017). *Model Based Testing-From requirements to tests.*
https://hdl.handle.net/10216/106157

Eclipse, (software) (2020).
https://en.wikipedia.org/wiki/Eclipse_(software). Ed-douibi,

H., Cánovas Izquierdo, J. L., & Cabot, J. (2017). Example-Driven Web API Specification Discovery. *Lecture Notes in Computer Science*, 267–284.
https://doi.org/10.1007/978-3-319-61482-3_16

Ed-douibi, H., Cánovas Izquierdo, J. L., & Cabot, J. (2018a). APIComposer: Data-Driven Composition of REST APIs. *Lecture Notes in Computer Science*, 161–169.
https://doi.org/10.1007/978-3-319-99819-0_12

Ed-douibi, H., Canovas Izquierdo, J. L., & Cabot, J. (2018b). Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach. 2018 *IEEE 22nd International Enterprise Distributed Object Computing Conference* (EDOC) (pp. 181-190). IEEE.
https://doi.org/10.1109/edoc.2018.00031

Fertig, T., & Braun, P. (2015). Model-driven Testing of RESTful APIs. *Proceedings of the 24th International Conference on World Wide Web*.
https://doi.org/10.1145/2740908.2743045

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. [Doctoral thesis University of California, Irvine].

Gao, J., Bai, X., & Tsai, W. T. (2011). Cloud testing-issues, challenges, needs and practice. *Software Engineering: An International Journal, 1*(1), 9-23.

Giessler, P., Gebhart, M., Sarancin, D., Steinegger, R., & Abeck, S. (2015). Best practices for the design of restful web services. *In International Conferences of Software Advances* (ICSEA) (pp. 392-397).

Graham, D., Van Veenendaal, E., & Evans, I. (2008). Foundations of Software Testing: ISTQB Certification. *Cengage Learning EMEA*, 30.

Izquierdo, J. L. C., & Cabot, J. (2014). Composing JSON-Based Web APIs. Composing JSON-based web APIs. *In ICWE 2014-14th International Conference on Web Engineering*, (Vol. 8541, pp. 390-399).
https://doi.org/10.1007/978-3-319-08245-5_24

Jackson, E. (2017). Combinatorial Testing in Cloud Computing.

Kuhn, D. R., Kacker, R. N., & Lei, Y. (2015). Measuring and specifying combinatorial coverage of test input configurations. *Innovations in Systems and Software Engineering, 12*(4), 249–261.
https://doi.org/10.1007/s11334-015-0266-2

Kuryazov, D., Winter, A., & Sandau, A. (2019). Sustainable Software Architecture for NEMo Mobility Platform. *Smart Cities/Smart Regions* – Technische, Wirtschaftliche Und Gesellschaftliche *Innovationen*, 229–239.
https://doi.org/10.1007/978-3-658-25210-6_18

Ma, S. P., Chen, Y. J., Syu, Y., Lin, H. J., & FanJiang, Y. Y. (2018). Test-Oriented RESTful Service Discovery with Semantic Interface Compatibility. *IEEE Transactions on Services Computing*, 1–1.
https://doi.org/10.1109/tsc.2018.2871133

Masse, M. (2011). *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.

Mulloy, B. (2013). Web API design.
https://hashingit.com/elements/research-resources/2012-web-api-design.pdf

Murphy, L., Alliyu, T., Macvean, A., Kery, M. B., & Myers, B. A. (2017). Preliminary Analysis of REST API Style Guidelines. *Ann Arbor, 1001*, 48109.

Neumann, A., Laranjeiro, N., & Bernardino, J. (2018). An analysis of public REST web service APIs. *IEEE Transactions on Services Computing.*
https://doi.org/10.1109/TSC.2018.2847344

Sangsanit, K., Kurutach, W., & Phoomvuthisarn, S. (2018). REST web service composition: A survey of automation and techniques. *2018 International Conference on Information Networking (ICOIN).* (pp. 116-121). IEEE.
https://doi.org/10.1109/icoin.2018.8343096

Sivanandan, S. (2014). Agile development cycle: Approach to design an effective Model Based Testing with Behaviour driven automation framework. *In 20th Annual International Conference on Advanced Computing and Communications (ADCOM)* (pp. 22-25). IEEE.
https://doi.org/10.1109/adcom.2014.7103243

Smart, J. (2014). *BDD in Action: Behavior-driven development for the whole software lifecycle*. Simon and Schuster.

Sneed, H. M., & Verhoef, C. (2015). Measuring test coverage of SoA services. *In 2015 IEEE 9th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments (MESOCA)* (pp. 59-66). IEEE.
https://doi.org/10.1109/MESOCA.2015.7328128

Surwase, V. (2016). REST API modeling languages-a developer's perspective. *International Journal of Science and Technology and Engineering, 2*(10), 634-637.

Utting, M., Pretschner, A., & Legeard, B. (2012). A taxonomy of model-based testing approaches. *Software testing, verification and reliability, 22*(5), 297-312.
https://doi.org/10.1002/stvr.456

Wolde, B. G., & Boltana, A. S. (2019). Combinatorial Testing Approach for Cloud Mobility Service. *In Proceedings of the 2019 2nd Artificial Intelligence and Cloud Computing Conference,* (pp. 6-13).
https://doi.org/10.1145/3375959