

A SYSTEM, METHOD AND COMPUTER READABLE MEDIUM FOR AUTOMATIC INTEGRATED TESTING OF APPLICATION PROGRAMMING INTERFACE USING SPREADSHEET-BASED PROGRAM

TECHNICAL FIELD

[0001] The present disclosure relates generally to a system, method and computer readable medium for automatic integrated testing of application programming interface. Embodiments of the disclosure are related to a system, and method and computer readable medium for automatic integrated testing of software programs using test cases based on spreadsheet-based programs.

BACKGROUND

[0002] Application Programming Interface (API) Testing is a type of testing that validates APIs. The purpose of API Testing is to check the functionality, reliability, performance, and security of the programming interfaces. In API Testing, instead of using standard user inputs (keyboard) and outputs, the testing framework sends calls to the API, gets output, and notes down the system's response.

[0003] Automated testing is a process undertaking test behavior by mechanical execution instead of human-driven action. In manual testing, after a test case is designed and the test case passes a review, the tester manually performs the test step by step according to the procedure described in the test case, and thereafter compares the actual result with the expected result. Manual testing is less efficient and, not only incurs manpower, but also greatly consumes time and hardware resources. Automated testing on the other hand saves manpower, time and hardware resources and improves test efficiency by eliminating the requirement of manual intervention throughout the testing process. Automated testing framework is a framework used for automated testing. The automated testing framework

normally consists of one or more automated test basic modules, automated test management modules, and automated test statistics modules.

[0004] Though automated testing is an improvement over manual testing, the conventional practices of automated testing also have certain drawbacks. In the traditional automated test frameworks, developers need to deal with a variety of complicated and cumbersome configurations, need to invest a lot of time to set up the framework, and also need professional language technology to design test cases according to the code, while the supported input data format is not rich. Additionally, in the conventional automated test frameworks, there are often many detours leading to instability and inefficiency.

SUMMARY

[0005] This specification describes a system, method and computer readable medium for automated testing of application programming interface. Embodiments of the disclosure are related to a system, and method and computer readable medium for automated testing of software programs using test cases based on spreadsheet-based programs.

[0006] The systems, methods and computer readable media disclosed in the present application are advantageous in several ways, including but not limited to, providing a complete set of automation services from data preparation to rule verification, supporting diverse and rich input data such as String, Object, Map, List and so on, conducting quick and efficient automated testing, thereby resulting in improved test execution efficiency, saving costs, time and manual effort employed by the user, significantly reducing the compatibility testing overhead and greatly reducing workload of compatibility testing.

[0007] It is, therefore, one aim of the disclosed embodiments to provide a system, method and computer readable medium for automatic integrated testing of APIs using spreadsheet-based programs, for example, Microsoft Excel.

[0008] It is, therefore, one aim of the disclosed embodiments to provide a system, method and computer readable medium that can pre-initialize the test data into the test environment database according to the prepared Structured Query Language (SQL) script.

[0009] It is, therefore, one aim of the disclosed embodiments to provide a system, method and computer readable medium that prepares a test case according to the test case template in a spreadsheet-based program. The system provides custom annotations and parsing converters in the spreadsheet-based program. The content is parsed into a test case object, which contains the header, request, and response validation rules required by the request. The interface request is issued according to the parameters and URL in the test case object, and the response returned by the interface is obtained, and then the validation is performed using response validator. The database checker verifies the response according to the check rule in the test case object.

[0010] It is, therefore, one aim of the disclosed embodiments to provide a system, method and computer readable medium for a complete set of automation services from data preparation to rule verification, supporting diverse and rich input data, such as string, object, map, list, and so on, can quickly and efficiently conduct automated testing. This can help the companies to improve the test execution efficiency, save time costs, and greatly reduce the workload of compatibility testing.

[0011] It is, therefore, one aim of the disclosed embodiments to provide a system, method and computer readable medium to develop custom annotation, initialize

SQL script on non-relational databases, assemble header and request parameter object and verify response and test case. Further, in some implementations, the system, method and computer readable medium use tool classes and validators, provide custom annotation used to convert a single case into an array, convert a single case into an array of attribute objects, and perform verification if object's status is enabled.

[0012] It is, therefore, one aim of the disclosed embodiments to provide a system, method and computer readable medium, comprising a testing unit, a test project and a database. The testing unit has a test case project and a common test project. The test case project comprises an initial data having at least one database script, wherein the database script is prepared and initialized by a user, at least one test case prepared by the user in a spreadsheet based program according to at least one test case template in the spreadsheet based program, and a test program customized by the user to read the database script and the test case. The common test project is developed and importable by the user in the test program. The common test project comprises at least one annotation, at least one tool, at least one domain, and at least one validator. The test project has at least one API to be tested. The database is accessed by the test program and the test project. The test program reads and parses the test case file, as at least one test case object or at least one text context object. The test case object is assembled with a header and at least one request parameter. An interface request from the test program is sent to the test project and where it is executed, and a response is sent to the test program. In some implementations, the header and request parameters are assembled using REST-Assured JAVA library and the interface request may be Rest-ful style. The test project validates the response against the test case and a test report is generated based on the validation result.

[0013] It is, therefore, one aim of the disclosed embodiments to provide a system, method and computer readable medium, comprising the steps of: developing and importing a common test project in a test program, preparing a test case in a test

case file based on at least one test case template in a spreadsheet based program, reading and parsing test case as at least one test case object, assembling a header and a request parameter in the test case object, issuing an interface request to a test project, receiving a response from the test project, verifying the response to produce at least one result and generating a report based on the result. In some implementations, the header and request parameters are assembled using REST-Assured JAVA library and the interface request may be Rest-ful style.

[0014] It is, therefore, one aim of the disclosed embodiments to provide a system, method and computer readable medium, wherein developing and importing a common test project comprises, introducing dependencies, developing custom annotation, classes and validators, preparing database utilities and pre-initializing at least one test data into a database according to a SQL script.

[0015] It is, therefore, one aim of the disclosed embodiments to provide a system, method and computer readable medium, wherein validating the response to produce at least one result comprises, receiving the response from test project; validating response header and response body of response with expected header and response of the test case, when the status code of both the response and the expected result are same, generating pass result when the response header and response body of response and expected header and response of the test case are same, checking error code of response and the expected result when the response header and response body of response and expected header and response of the test case are not same, generating fail result when the response header and response body of response and expected header and response of the test case are not same and generating the report based on the pass result or fail result. The test case is converted to an array of attribute objects. The request is sent to the test projects only when the object status is enabled.

[0016] It is, therefore, one aim of the disclosed embodiments to provide a system, method and computer readable medium, wherein generating the report based on

the pass result or fail result, further comprise: querying at least one value from the database; receiving at least one database result from the database; comparing the database result and response; obtaining pass or fail result; and generating the report based on pass or fail result.

[0017] In a first aspect of the present disclosure, a system for automatic integrated testing of an application programming interface (API) is disclosed. The system comprises a test framework comprising a test case project, a common test project, a test project having the API to be tested and a database, such that the test case project comprises: at least one database script prepared and initialized by a user; at least one test case written in a test case file according to at least one test case template in a spreadsheet-based program; and a test program configured to read the database script and the test case; and the common test project is developed and configured to be imported in the test program.

[0018] According to an embodiment in conjunction to the first aspect of the present disclosure, the common test project comprises at least one annotation, at least one tool, at least one domain, and at least one validator.

[0019] According to an embodiment in conjunction to the first aspect of the present disclosure, the test program reads and parses the test case as at least one test case object or at least one test context object.

[0020] According to an embodiment in conjunction to the first aspect of the present disclosure, the test case object is assembled with a header and at least one request parameter.

[0021] According to an embodiment in conjunction to the first aspect of the present disclosure, the test project receives an interface request from the test program.

[0022] According to an embodiment in conjunction to the first aspect of the present disclosure, the test case object is assembled through a REST-Assured library.

[0023] According to an embodiment in conjunction to the first aspect of the present disclosure, the request is a RESTful request.

[0024] According to an embodiment in conjunction to the first aspect of the present disclosure, the test program receives a response to the interface request from the test project.

[0025] According to an embodiment in conjunction to the first aspect of the present disclosure, the test case supports request or response of multiple data formats.

[0026] According to an embodiment in conjunction to the first aspect of the present disclosure, the test project validates the response with an expected response in the test case and a test report is generated based on a result of the validation.

[0027] In a second aspect of the present disclosure, a method for automatic integrated testing of an application programming interface (API) is disclosed. The method comprising: developing and importing a common test project in a test program; preparing a test case in a test case file based on at least one executable test case template in a spreadsheet-based program; reading and parsing, by the test program, the test case as at least one test case object or at least one text context object; assembling a header and a request parameter in the test case object; issuing, by the test program, an interface request to a test project; receiving, by the test program, a response from the test project; validating, by the test program, the response with an expected response in the test case; and generating, by the test program, a report based on a result of the validation.

- [0028]** According to an embodiment in conjunction to the second aspect of the present disclosure, assembling of the test case object is done through a REST-Assured JAVA library.
- [0029]** According to an embodiment in conjunction to the second aspect of the present disclosure, the interface request is a RESTful request.
- [0030]** According to an embodiment in conjunction to the second aspect of the present disclosure, developing and importing the common test project comprises: introducing at least one dependency; and developing at least one custom annotation, at least one class and at least one validator and preparing at least one database utility.
- [0031]** According to an embodiment in conjunction to the second aspect of the present disclosure, the method further comprises pre-initializing at least one test data into a database according to a SQL script.
- [0032]** According to an embodiment in conjunction to the second aspect of the present disclosure, wherein validating the response to produce at least one result comprises: comparing, by a response validator, a test object status code of the received response with an expected test object status code of the test case.
- [0033]** According to an embodiment in conjunction to the second aspect of the present disclosure, comparing by the response validator comprises: comparing an error code of the received response with an expected error code of the test case.
- [0034]** According to an embodiment in conjunction to the second aspect of the present disclosure, the method further comprises validating, by the response validator, the response header and response body of response with an expected header and response of the test case.

[0035] According to an embodiment in conjunction to the second aspect of the present disclosure, the method further comprises querying, by a database validator, at least one value from the database; receiving, by the database validator, at least one database result from the database; and comparing the database result and an expected database result of the test case.

[0036] According to an embodiment in conjunction to the second aspect of the present disclosure, the test case is converted to an array of attribute objects.

[0037] According to an embodiment in conjunction to the second aspect of the present disclosure, the interface request is sent to the test project only when a status of the object is enabled.

[0038] In a third aspect of the present disclosure, a computer readable medium storing a program comprising instructions that, when executed by a system, cause the system to perform the method for automatic integrated testing of an application programming interface (API) is disclosed. The instructions, when executed by the system comprise, developing and importing a common test project in a test program; preparing a test case in a test case file based on at least one executable test case template in a spreadsheet-based program; reading and parsing, by the test program, the test case as at least one test case object or at least one text context object; assembling a header and a request parameter in the test case object; issuing, by the test program, an interface request to a test project; receiving, by the test program, a response from the test project; validating, by the test program, the response with an expected response in the test case; and generating, by the test program, a report based on a result of the validation.

[0039] According to an embodiment in conjunction to the third aspect of the present disclosure, assembling of the test case object is done through a REST-Assured JAVA library.

- [0040]** According to an embodiment in conjunction to the third aspect of the present disclosure, the interface request is a RESTful request.
- [0041]** According to an embodiment in conjunction to the third aspect of the present disclosure, developing and importing the common test project comprises: introducing at least one dependency; and developing at least one custom annotation, at least one class and at least one validator and preparing at least one database utility.
- [0042]** According to an embodiment in conjunction to the third aspect of the present disclosure, the instructions when executed by the system further comprise, pre-initializing at least one test data into a database according to a SQL script.
- [0043]** According to an embodiment in conjunction to the third aspect of the present disclosure, validating the response to produce at least one result comprises: comparing, by a response validator, a test object status code of the received response with an expected test object status code of the test case.
- [0044]** According to an embodiment in conjunction to the third aspect of the present disclosure, the instructions when executed by the system further comprise, querying, by a database validator, at least one value from the database; receiving, by the database validator, at least one database result from the database; and comparing the database result and an expected database result of the test case.
- [0045]** According to an embodiment in conjunction to the third aspect of the present disclosure, the test case is converted to an array of attribute objects.
- [0046]** According to an embodiment in conjunction to the third aspect of the present disclosure, the interface request is sent to the test project only when a status of the object is enabled.

[0047] The following disclosure is provided to facilitate an understanding of some of the innovative features unique to the disclosed embodiment and is not intended to be a full description. A full appreciation of the various aspects of the embodiments disclosed herein can be gained by taking into consideration the entire specification, claims, drawings, and abstract as a whole.

BRIEF DESCRIPTION OF THE DRAWINGS

[0048] **FIG. 1** is a block diagram depicting an example automated testing system **100**, in accordance with the disclosed embodiments;

[0049] **FIG. 2** is a diagrammatic representation of an example list of common test projects **200**, in accordance with the disclosed embodiments;

[0050] **FIG. 3** is a flow diagram that illustrates an example automated testing method **300**, in accordance with the disclosed embodiments;

[0051] **FIG. 4** is a flow diagram that illustrates another example automated testing method **400**, in accordance with the disclosed embodiments; and

[0052] **FIG. 5** is a flow diagram that illustrates an example validation method **500** of a received response, in accordance with the disclosed embodiments.

[0053] The summary above, as well as the following detailed description of illustrative embodiments, is better understood when read in conjunction with the appended drawings. For the purpose of illustrating the present disclosure, exemplary constructions of the disclosure are shown in the drawings. However, the disclosure is not limited to specific methods and instrumentalities disclosed herein. Moreover, those in the art will understand that the drawings are not to scale. Wherever possible, like elements have been indicated by identical numbers.

DETAILED DESCRIPTION OF ILLUSTRATIVE EMBODIMENTS

[0054] The particular configurations discussed in the following description are non-limiting examples that can be varied and are cited merely to illustrate at least one embodiment and are not intended to limit the scope thereof.

[0055] **FIG. 1** is a block diagram depicting an example automated integrated testing system **100**, in accordance with the disclosed embodiments. The system **100** mainly comprises a test framework **102**, a database **114** and a test project **116**.

[0056] The test framework **102** may adopt frameworks such as Poor Obfuscation Implementation (POI), TestNG, REST-Assured, PowerMock, JUnit and so on. Apache POI is an API that allows programmers to create, modify, and display MS Office files using Java programs. It is an open source library to design or modify Microsoft Office files using Java programs. TestNG is a testing framework with features that make it suitable for use in regression test automation projects. Data Provider is one such feature in TestNG. It allows a test method to be executed with multiple sets of data. Rest-Assured is a Java-based library that is used to test RESTful Web Services. This library behaves like a headless client to access Representational State Transfer (REST) web services. Rest-Assured library also provides the ability to validate the HTTP Responses received from the server. For example, the user can verify the Status code, Status message, Headers and even the Body of the response. This makes Rest-Assured a very flexible library that can be used for testing. PowerMock is a framework that extends other mock libraries such as EasyMock with more powerful capabilities. PowerMock uses a custom classloader and bytecode manipulation to enable mocking of static methods, constructors, final classes and methods, private methods, removal of static initializers and more. By using a custom classloader no changes need to be done to the Integrated Development Environment (IDE) or continuous integration

servers which simplifies adoption. In some implementations, the testing framework **102** is developed using JAVA.

[0057] The test framework **102** comprises a common test project **104** and a test case project **106**. The common test project **104** is developed and importable by the test case project **106**. The common test project **104** includes but is not limited to, one or more of the following:

1. Annotations;
2. Tools;
3. Domains; and
4. Validators

[0058] The test case project **106** comprises a database script **108**, at least one test case file **112** and test program **110**. The test program **110** is customized by the user to read the database script **108** and the test case **112**, assisted by the imported common test project **104** in this regard. The database script **108** and the test case file **112** are prepared by the user or programmer based on the testing requirements. The database script **108** is a series of command line scripts pertaining to operation and function of the database **114**. The database script **108** has at least one database script of an initial data prepared and initialized by a user or a programmer. Initial data is prepared by the user if the test case requires a special test data for testing. For example, if the test case is to test if personnel from organization A can access a source belonging to organization B, relevant data pertaining to organizations A and B is prepared by the user or programmer and stored in the database **114**. In some implementations, the user or programmer may write a database script to clear the initial data after a test case execution. If the user does not write said script, the data remains in the database after the test case execution. It should be noted that, it is not mandatory to have at least one database script **108** in the test case project **106**, it depends on whether the database **114** needs pre-initialization or not.

[0059] In some implementations, test data may be pre-initialized into the database **114** according to a prepared SQL script. In addition to initializing the traditional relational database script, the test program **110** also supports data initialization **132** operations of non-relational databases **114**. For example, MongoDB can initialize the database script through MongoDBUtils, and then iterate through each case in the test case file **112**.

[0060] The test case file **112** is prepared by the user in a spreadsheet-based program, for example, Microsoft Excel. Each line in the spreadsheet-based program of test case file **112** represents an individual test case. Keys are used in the header column that enable the test program **110** to read the test case file **112** and parse each column into attributes of test case objects. The following table discloses examples of keys that may be used in the test case file **112** along with their meanings:

TABLE 1

Key	Meaning
Description	Test case description
REQ_B	request body expressed as a path file, for example a json path. E.g. user.name=abc
REQ_P	request path variable
REQ_H	request header
REQ_Q	request query parameter
RES_B	response body expressed as path file, for example

	a json path
RES_H	response header
HttpStatus	http status
ErrCode	error code constant ordinal
DB_CHECK	verify database result

[0061] The test case file **112** supports request and response of multiple data formats such as String, Object, List, Map and so on. In some implementations, when the test program **110** runs the test case corresponding to each test case file **112**, it initializes a prepared SQL script to the database **114**. In addition to initializing the traditional relational database script, the framework also supports data initialization operations of non-relational databases. For example, MongoDB can initialize the database script through MongoDBUtils, and then iterate through each Case in the test case file **112**. A custom annotation “@TestCaseFile” is developed to mark the text case file **112**. The content of the test case file **112** is read and parsed by the test program **110** as test case (TestCase) objects and test context (TextContext) objects. Test case objects describe all the necessary information constructing a test case, for example, test case number, name, description, request information, response information, expected result or error. Test context objects encapsulate the context in which a test is executed, agnostic of the actual testing framework in use and refer to the data required to run a test case, for example, locale information, request Internet Protocol (IP) address and so on. The test case objects may include but not limited to, request body (requestBody), request header (requestHeader), request query parameter (requestQueryParam), response header (responseHeader), response body (responseBody), HTTP status (HttpStatus), error code (errorCode), database result (dbResult), and so on. An example of test case file **112** is of the format

“@TestCaseFile("categories/addCategoryCase.xlsx)"). The test project **116** has at least one API to be tested.

[0062] The database **114** is accessed by the test program **110** and the test project **116** during the initialization, validation and execution of test case file **112**. The test program **110** executes the test case by reading the data in the test case file **112** and parsing the data to test case objects and test context objects. The test program **110** then forms and sends a test request **122** to the test project **116**. In some implementations, the testing framework **102** comprises a REST-Assured framework and the test request **122** sent to the test project **116** is a Restful-style interface request. To test the API of the test project **116**, the test project **116** with the help of the database **114** executes the test case contained in the test case file **112** and produces a test response **124**. The test project **116** will initiate a connection to the database **114**, and send a query **126** to the database **114** with specific SQL statement, for example, “select * from uum_user where user_id = ‘test_user’”. In some implementations, the connection with the database is initiated by the test project via Java Database Connectivity (JDBC). The database **114** returns a selected result set **128** back to the test project **116** in the form of a result set (ResultSet). Result set in the test project **116** refers to a response from the database **114**. For example, assuming that, an API “post /v1/system/users” creates a new user in the database **114**, the response **124** of the API of the test project **116** may contain a status “201” and a new “user id”, as a response body. The developer might want to check in the database **114** and confirm whether the new user is really inserted or not in the database **114**. Hence, a database check may be configured in the test case file **112** and when the test case is executed, the test project **116** will query **126** the database **114** and get the result set (ResultSet) **128** to compare with the data in the test case file **112**. The test project **116** will analyze and parse the resultset **128** to a specific format, for example, java objects, to compare with an expected data defined in the test case file **112**.

[0063] In an example wherein the API of the test project **116** is “POST/categories”, the test case objects may be requestHeader and requestBody attributes after parsing. The test program **110** may utilize RestAssured to obtain the response:

```
import static io.restassured.RestAssured.given;
Response response =
    given().headers(HeaderBuilder.build(testCase.getRequestHeader()))
    .and().body(testCase.getRequestBody())
    .post("/categories").then().extract().response();
```

[0064] Other methods such as POST, GET, PUT, PATCH, DELETE may also be tested in some implementations.

[0065] Once test program **110** receives the test response **124** from the test project **116**, the test program **110** uses validators such as response validator (ResponseValidator) and database validator (DataBaseValidator) to compare and verify the received test response **124** and the test case. The response validator verifies that the returned status, returned header, and the returned value in the received test response **124** is consistent with the expected return result of the test case designed in the test case file **112**. The database validator sends a query **118** to the database **114** regarding the database result expected by the test case designed in the test case file **112** and receives a response **120** from the database **114**. The test program **110** compares and validates the received database result with the expected database result. Once the execution of the test case and validation of the received response is complete, the test program **110** creates a test report of the visualized automated test with a pass or fail use case, test steps, test instructions and code.

[0066] A HTTP request normally contains a HTTP URL, for example: “http://120.0.0.1/v1/system/users”, a request header, for example, a user ID and timestamp and a request body, for example in JSON (JavaScript Object Notation) format. A HTTP response normally contains a HTTP response header, and a

response body, for example in JSON format. The response body is parsed to objects, for example JAVA objects, for validation. The database is queried with the response, for example, response ID returned as part of the response body received from the test project **116**, to obtain the database data. The database query result is validated by the database validator against the expected database data defined in the test data file **112**.

[0067] **FIG. 2** is a diagrammatic representation of an example common test project **200**, in accordance with the disclosed embodiments. The common test project **200** comprises at least one annotation **202**, at least one tool **204**, at least one domain **206**, and at least one validator **208**. Annotation **202** is a tag that represents the *metadata* that is, attached with class, interface, methods or fields to indicate some additional information of a file. As disclosed above, custom annotations **202** may be developed in the test framework such as `@TestCaseFile`. Tools **204** may comprise tool classes including but not limited to `TestCaseParser`, `SQLParser`, `MongoDBUtils`, `DBCheckParser` and so on. The SQL Parser parses a SQL query in a string field. When parsing a query, the processor generates fields based on the fields defined in the SQL query and specifies the CRUD operation, table, and schema information in record header attributes. `MongoDBUtils` refers to a helper class featuring helper methods for internal `MongoDb` classes mainly intended for internal use within the framework. Such self-developed parsers are used to read and interpret the symbols and data from a spreadsheet-based program. The purpose of a `TestCaseParser` parser is to convert the array of attribute objects obtained from the custom annotation `@TestCaseFile` into a `TestCase` object. Each `TestCase` object is verified to check if the object's status is enabled. However, if the object's status is found to be disabled, the corresponding interface request is skipped, and the parser continues to the next case. Domains **206** are common classes/ objects which are not bound by any special restriction other than those forced by the Programming language specification and do not require any class path. In an example where JAVA programming language is used, the objects may be Plain Old Java Objects (POJO), as generally used in Java which are not bound

by any special restriction other than those forced by the Java Language Specification and do not require any class path. In the common test project **200**, two main domains are used such as TestCase and TestContext. TestCase domain describes all the necessary information, which is required to construct a test case, for example, test case number, name, description, request information, response information, expected result or error. Whereas TestContext domain holds all the data required to run a test case, for example, locale information, request IP address and so on.

[0068] Validators **208** are used to verify a test response received as a result of executing a test case for a specific API of a test project based on an interface request. The received test response is validated by means of validators **208** to compare and verify the received test response and the expected result of the test case.

[0069] For example, a generic validator “GenericValidator” validates the received test response generally based on a validatorTag (e.g. @NotNull). Thus, if the test response in the test case file is tagged as @NotNull, the generic validator will use assertNull method to validate. In another example, a response validator “ResponseValidator ” validates that the received test response’s status code, headers, body, and error code (in desired response is “error”) match with the expected response of the test case designed in the test case file. The response validator first determines whether the response’s status code is the same as TestCase’s HTTPStatus. The status code may be OK, NO_CONTENT CREATED, or ACCEPTED. If affirmative, the response validator continues to validate if the test response headers and test response body are consistent with that of the expected response designed in the test case file. In case the desired response is “error”, the response validator validates if the test response’s error code is the same as the expected error response designed in the test case file. In another example, a database validator “DatabaseValidator” validates the database execution performed based on test cases. The database validator queries the

value in a database and the database result expected by the test case designed in the spreadsheet. The database validator first reads the expected response designed in the test case file and then validates the received test response. An example of an expected database result is presented below:

DB CHECK:uum_role
Q(role_name=Org2Admin updated_date=@Now)==2

[0070] In the above example, the expected result design in the test case file for the database validator is to “check table ‘uum_role’, the count of record whose role_name is “Org2Admin” and updated_date is within past 5 mins (annotation @Now) is “2”. The database validator compares the received test result from the database and determines whether the count matches with the defined expected response in the test case file.

[0071] **FIG. 3** is a flow diagram that illustrates an example automated testing method **300**, in accordance with the disclosed embodiments. As disclosed in **FIG. 3**, the automated testing method **300** can be implemented, for example, at step **302** by introducing in a test framework, dependencies, developing custom annotations, classes, tools, validators and preparing database utilities. The test framework may adopt frameworks such as Poor Obfuscation Implementation (POI), TestNG, REST-Assured, PowerMock, JUnit and so on. Apache POI is an API that allows programmers to create, modify, and display MS Office files using Java programs. TestNG is a testing framework with features that make it suitable for use in regression test automation projects. Rest-Assured is a Java-based library that is used to test RESTful Web Services. PowerMock is a framework that extends other mock libraries such as EasyMock with more powerful capabilities. In some implementations, the test framework is developed using JAVA. Custom annotations, such as “@TestCaseFile” may be developed in the test framework. Further, tools including but not limited to TestCaseParser, SQLParser, MongoDBUtils, DBCheckParser and so on are also introduced in the test

framework. Such self-developed parsers are used to read and interpret the symbols and data from a spreadsheet-based program. The purpose of a TestCaseParser parser is to convert the array of attribute objects obtained from the custom annotation @TestCaseFile into a TestCase object. Each TestCase object is verified to check if the object's status is enabled. However, if the object's status is found to be disabled, the corresponding interface request is skipped, and the parser continues to the next case. Validators are used by the test framework to verify a test response received as a result of executing a test case for a specific API of a test project based on an interface request. The received test response is validated by means of validators to compare and verify the received test response and the expected result of the test case.

[0072] As disclosed in **FIG. 3**, step **302** also includes preparation of database utilities including initial data and database scripts. Initial data may be prepared by a user if the test case requires a special test data for testing. For example, if the test case is to test if personnel from organization A can access a source belonging to organization B, relevant data pertaining to organizations A and B is prepared by the user or programmer and stored in a database. Database scripts are a series of command line scripts pertaining to operation and function of the database. In some implementations, the user or programmer may write a database script to clear the initial data after a test case execution. If the user does not write said script, the data remains in the database after the test case execution. For example, "dbUtils.initDB("delete_rms.sql")" may be used to execute database scripts. In some implementations, test data may be pre-initialized into the database according to a prepared SQL script. In addition to initializing the traditional relational database script, the framework also supports data initialization operations of non-relational databases. For example, MongoDB can initialize the database script through MongoDBUtils, and then iterate through each Case in the test case file.

[0073] Thereafter, at step **304**, the user provides a test case file based on a spreadsheet-based program. The test case file based on the spreadsheet-based

program is marked with the custom annotation `@TestCaseFile`. Each line in the spreadsheet-based program of the test case file represents an individual test case and keys are used in the header columns to identify the content. For example, “Description” key refers to a test case description, “REQ_B” key represents a request body expressed as a path file, example “user.name=abc”, optionally a JSON path, “REQ_P” key represents a request path variable, “REQ_H” key represents a request header, “REQ_Q” key represents a request query parameter, “RES_B” key represents a response body expressed as a path file, for example, a JSON path, “RES_H” key represents a response header, “HttpStatus” key represents a HTTP status, “ErrCode” key represents an error code constant ordinal, “DB_CHECK” key represent verify database result.

[0074] Thereafter, the example method **300** is implemented by a test program of the test framework reading the content of the test case file and parsing the content as test case (TestCase) objects and test context (TextContext) objects, at step **306**. Test case objects describe all the necessary information constructing a test case, for example, test case number, name, description, request information, response information, expected result or error. Test context objects encapsulate the context in which a test is executed, agnostic of the actual testing framework in use and refer to the data required to run a test case, for example, locale information, request IP address and so on. The test case objects may contain, including but not limited to, request body (requestBody), request header (requestHeader), request query parameter (requestQueryParam), response header (responseHeader), response body (responseBody), HTTP status (httpStatus), error code (errorCode), database result (dbResult), and so on. An example of a test case file is of the format “@TestCaseFile("categories/addCategoryCase.xlsx)”.

[0075] Thereafter, at step **308**, the test framework assembles the Header and Request parameters in the test case objects. In an example, the test framework assembles the Header and Request parameters in the test case objects through

REST-Assured JAVA library. At step **310**, the test framework issues an interface request to a test project containing at least one API to be tested. In the above example wherein header and request parameters are assembled using REST-Assured JAVA library, the interface request may be Rest-ful style. In response to the interface request, at step **312**, the test framework receives a response from the API. In an example wherein the API of the test project is “POST/categories”, the test case objects may be requestHeader and requestBody attributes after parsing. The test framework may utilize RestAssured to obtain the response:

```
import static io.restassured.RestAssured.given;
Response response =
    given().headers(HeaderBuilder.build(testCase.getRequestHeader()))
    .and().body(testCase.getRequestBody())
    .post("/categories").then().extract().response();
```

[0076] Other methods such as POST, GET, PUT, PATCH, DELETE may also be tested in some implementations.

[0077] On receiving the test response from the test project, the test framework compares and validates the received response with the test case by way of response validator (ResponseValidator) and database validator (DataBaseValidator) at step **314**. The response validator verifies that the returned status, returned header, and the returned value in the received test response is consistent with the expected return result of the test case designed in the test case file. The database validator sends a query to the database regarding the database result expected by the test case designed in the test case file. On conclusion of the execution of the test case and validation of the received response, the method **300** can be implemented, for example, at step **316**, a test report is generated by the framework of the visualized automated test with a pass or fail use case, test steps, test instructions and code.

[0078] FIG. 4 is a flow diagram that illustrates another example automated testing method **400**, in accordance with the disclosed embodiments. As disclosed in FIG. 4, the example method **400** can be implemented by a test program, by reading of a test framework, a test case with a custom annotation @TestCaseFile and parsing the test case by the test program using a parser “TestCaseParser” to an array of attribute objects including but not limited to test case “TestCase” objects and text context “TestContext” objects at step **402**. At step **404**, the method **400** is implemented by the test program, by checking whether the status of the test case object is enabled or not. If the test case object status is enabled, the method is implemented by the test program, by sending an interface request to a test project containing an API to be tested, at step **406**.

[0079] On receiving a response from the test project, at step **408**, the method **400** is implemented by the test program, by comparing and validating the received test response, with the test case, at step **410**. The validating is conducted by the test program by a validator, for example, a response validator (ResponseValidator) or a database validator (DataBaseValidator). The response validator verifies that the returned status, returned header, and the returned value in the received test response is consistent with the expected return result of the test case designed in the test case file. The database validator sends a query to the database regarding the database result expected by the test case designed in the test case file. On conclusion of the execution of the test case, the method **400** can be implemented by the test program, by generating a test report of the visualized automated test with a pass or fail use case, test steps, test instructions and code at step **412**.

[0080] If at step **404**, checking by the test program of the test case object reveals that the test case object is disabled, the test program skips the interface request of the current test case and continues to the next test case, at step **414**. At step **416**, the method **400** is implemented by the test program, by checking whether the status of the next test case object is enabled or not. If the test case object status of the next test case is enabled, the method **400** is implemented by the test

program, by executing the step **406**. If the test case object status of the next test case is disabled, the method **400** is implemented by the test program, by executing the step **414**.

[0081] **FIG. 5** is a flow diagram that illustrates an example validation method **500** of a received test response, in accordance with the disclosed embodiments. As disclosed in **FIG. 5**, the method **500** can be implemented for example by receiving, by a test program, a test response to a test case from a test project at step **502**. At step **504**, a response validator (ResponseValidator) verifies that a test object status code of the received test response is the same as an expected test object status code (TestCase's HTTPStatus) of the test case designed in the test case file. In case the status code is "error", the response validator validates if the received test response's error code is the same as the expected error response designed in the test case file. If the code is the same at step **506**, the response validator (ResponseValidator) verifies that the response header and response body of the received test response is the same as an expected response header and response body of the test case designed in the test case file at step **508**. If the response header and response body is the same at step **510**, a database validator "DatabaseValidator" sends a query to a database regarding a value of a database result expected by the test case designed in the test case file at step **512**. The database validator receives the database result from the database at step **514**. The test program compares and validates the received database result with the expected database result at step **516**. Once the execution of the test case and validation of the received response is complete, the test program generates a test report of the visualized automated test with a passed test case, test steps, test instructions and code at step **518**. An example passed report is as below:

Passed

com.oneconnect.sg.inttest.ReportTemplate.PutReportTemplateByIdTest.testSuccess (from TestSuite)

[0082] If the status code is not the same at step **506**, the method **500** is implemented by generating a test report of failed test case, test steps, test instructions and code at step **518**. If the response header and response body is not the same at step **510**, the method **500** is implemented by generating a test report of failed test case, test steps, test instructions and code at step **518**. An example failed report is as below:

```

Regression
com.oneconnect.sg.inttest.ReportTemplate.PutReportTemplateByIdTest.testSuccess (from TestSuite)

Error Message
Error http status code. expected [200] but found [422]

Stacktrace
java.lang.AssertionError: Error http status code. expected [200] but found [422]
    at com.oneconnect.sg.inttest.ReportTemplate.PutReportTemplateByIdTest.testSuccess(PutReportTemplateByIdTest.java:48)

Standard Output
2020-01-17 06:23:53.876 INFO 13863 --- [main] trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework
[org.springframework.cloud.autoconfigure.ConfigurationPropertiesRebinderAutoConfiguration$$EnhancerBySpringCGLIB$$97575c0] is ..
eligible for auto-proxying)

:: Spring Boot :: (v2.1.0.RELEASE)

```

[0083] The systems, methods and computer readable media disclosed in the present application are advantageous in several ways, including but not limited to, providing a complete set of automation services from data preparation to rule verification, supporting diverse and rich input data such as String, Object, Map, List and so on, conducting quick and efficient automated testing, thereby resulting in improved test execution efficiency, saving costs, time and manual effort

employed by the user, significantly reducing the compatibility testing overhead and greatly reducing workload of compatibility testing.

[0084] It will be appreciated that variations of the above disclosed and other features and functions, or alternatives thereof, may be desirably combined into many other different systems or applications. Also, various presently unforeseen or unanticipated alternatives, modifications, variations or improvements therein may be subsequently made by those skilled in the art which are also intended to be encompassed by the following claims.

[0085] Although embodiments of the current disclosure have been described comprehensively in considerable detail to cover the possible aspects, those skilled in the art would recognize that other versions of the disclosure are also possible.

CLAIMS

What is claimed is:

1. A system for automatic integrated testing of an application programming interface (API), comprising:
 - a test framework comprising a test case project and a common test project;
 - a test project having the API to be tested; and a database,
 - such that the test case project comprises:
 - at least one database script prepared and initialized by a user;
 - at least one test case written in a test case file according to at least one test case template in a spreadsheet-based program;
 - and
 - a test program configured to read the database script and the test case; and
 - the common test project is developed and configured to be imported in the test program.
2. The system of claim 1, wherein the common test project comprises at least one annotation, at least one tool, at least one domain, and at least one validator.
3. The system of claims 1 or 2, wherein the test program reads and parses the test case as at least one test case object or at least one test context object.
4. The system of claim 3, wherein the test case object is assembled with a header and at least one request parameter.
5. The system of claims 1 to 4, wherein the test project receives an interface request from the test program.

6. The system of claim 5, wherein the test case object is assembled through a REST-Assured library and the interface request is a RESTful request.
7. The system of claim 6, wherein the test program receives a response to the interface request from the test project.
8. The system of claim 7, wherein the test case supports request or response of multiple data formats.
9. The system of claims 7 or 8, wherein the test project validates the response with an expected response in the test case and a test report is generated based on a result of the validation.
10. A method for automatic integrated testing of an application programming interface (API), comprising:
 - developing and importing a common test project in a test program;
 - preparing a test case in a test case file based on at least one executable test case template in a spreadsheet-based program;
 - reading and parsing, by the test program, the test case as at least one test case object or at least one text context object;
 - assembling a header and a request parameter in the test case object;
 - issuing, by the test program, an interface request to a test project;
 - receiving, by the test program, a response from the test project;
 - validating, by the test program, the response with an expected response in the test case; and
 - generating, by the test program, a report based on a result of the validation.

11. The method of claim 10, wherein assembling of the test case object is done through a REST-Assured JAVA library and the interface request is a RESTful request.
12. The method of claim 10, wherein developing and importing the common test project comprises:
 - introducing at least one dependency; and
 - developing at least one custom annotation, at least one class and at least one validator and preparing at least one database utility.
13. The method of claim 12 further comprising:
 - pre-initializing at least one test data into a database according to a SQL script.
14. The method of claims 10 to 13, wherein validating the response to produce at least one result comprises:
 - comparing, by a response validator, a test object status code of the received response with an expected test object status code of the test case.
15. The method of claim 14, wherein comparing by the response validator comprises:
 - comparing an error code of the received response with an expected error code of the test case.
16. The method of claims 14 or 15, further comprising:
 - validating, by the response validator, the response header and response body of response with an expected header and response of the test case.
17. The method of claim 16, further comprising:

querying, by a database validator, at least one value from the database;
receiving, by the database validator, at least one database result from the database; and
comparing the database result and an expected database result of the test case.

18. The method of claim 10, wherein the test case is converted to an array of attribute objects.

19. The method of claim 18, wherein the interface request is sent to the test project only when a status of the object is enabled.

20. A computer readable medium storing a program comprising instructions that, when executed by a system, cause the system to perform the method for automatic integrated testing of an application programming interface (API) of any of claims 10 to 19.