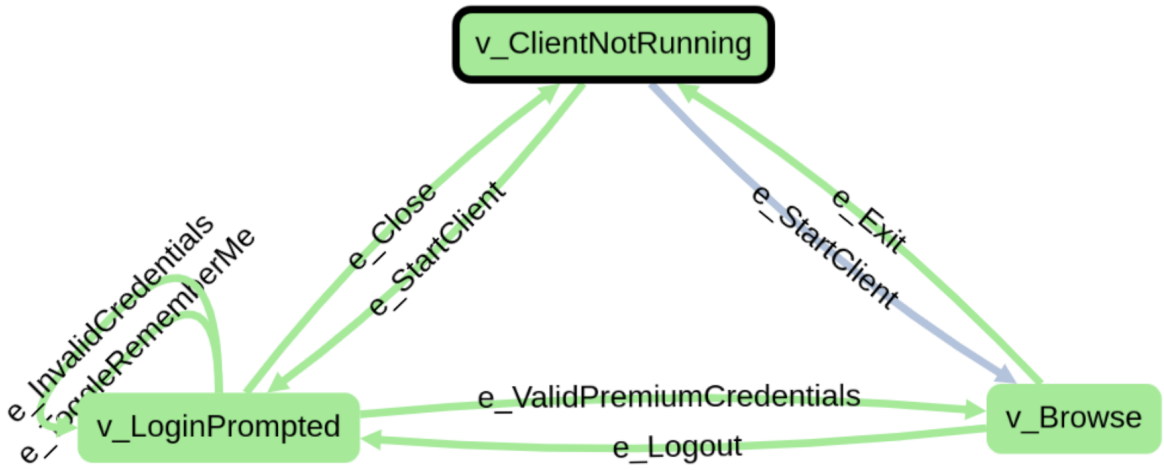
Model-Based Testing using GraphWalker and Java

Overview

GraphWalker is a tool that addresses State Transition Model-Based Testing; in other words, it allows you to perform modeling around states and transitions between those states using directed graphs.



Here is some clarification around some key concepts using the information provided by GraphWalker's documentation that explains them clearly:

- **edge**: *An edge represents an action, a transition**.** An action could be an API call, a button click, a timeout, etc. Anything that moves your System Under Test into a new state that you want to verify. But remember, there is no verification going on in the edge. That happens only in the vertex.*
- **vertex:** *A vertex represents verification, an assertion. A verification is where you would have assertions in your code. It is here that you verify that an API call returns the correct values, that a button click actually did close a dialog, or that when the timeout should have occurred, the System Under Test triggered the expected event.*
- **model**: *A model is a graph, which is a set of vertices and edges.*

*From a model, GraphWalker will generate a **path** through it. A model has a **start element**, and a **generator** which rules how the path is generated, and associated **stop condition** which tells GraphWalker when to stop generating the path.*

Generators and stop conditions are essential in GraphWalker (more info here and here), as they influence how the model will be "walked" and until when.

Multiple models can interact with one another (i.e. jump from one to other and vice-versa), using shared states (i.e. vertices that have a "shared name").

Each model has an internal state with some variables - its **context**. Besides, and since GraphWalker can transverse multiple models, there is also a **global context**.

We can also add actions and guards to the model, which can affect how the model is walked and how it behaves:

- **action**: a way of setting variables in the model or global context; actions are implemented using JavaScript
- **guard**: a way of blocking/guard edges from being walked/executed, usually considering variables stored in the model or global context; guards are implemented using JavaScript.

In sum, we model (i.e. build a model) a certain aspect related to our system using directed graphs; the model represents a test idea that describes expected behaviors. Checks are implemented in the vertices (i.e. states) and actions are performed in the edges. GraphWalker will then "walk" the model (i.e. perform a set of "steps"/edges) using a generated path. While doing so, it looks at JavaScript guards to check if edges can be "walked" and performs JavaScript based *actions* to set internal context variables . It stops "walking" if stop condition(s) are met.

To build the model, we can use a visual tool and (GraphWalker Studio) and export it to a JSON file.

Mapping concepts to Xray

Tests

Besides other entities, in Xray we have Test issues and "requirements" (i.e. issues that can be covered with Tests).

In GraphWalker, the testing is performed continuously by walking a path (as a result of its generator) and until certain condition(s) is(are) met.

This is a bit different from traditional, sequential test scripts where each one has a set of well-defined actions and expected results.

We can say that GraphWalker produces dynamic test cases, where each one corresponds to the full path that was generated. Since the number of possible paths can be quite high, we can follow a more straightforward approach: consider each model a Test, no matter exactly what path is executed. Remember that a model in itself is a high-level test idea, something that you want to validate; therefore, this seems a good fit as long as we have the means to later on debug it.

Requirements

What about "requirements"?

Well, even though GraphWalker allows you to assign one or more requirement identifiers to each vertex, it may not be the most suitable approach linking our model (or parts of it) to requirements. Therefore, and since we consider the model as a Test, we can eventually link each model to a "requirement" later on in Jira.

Results

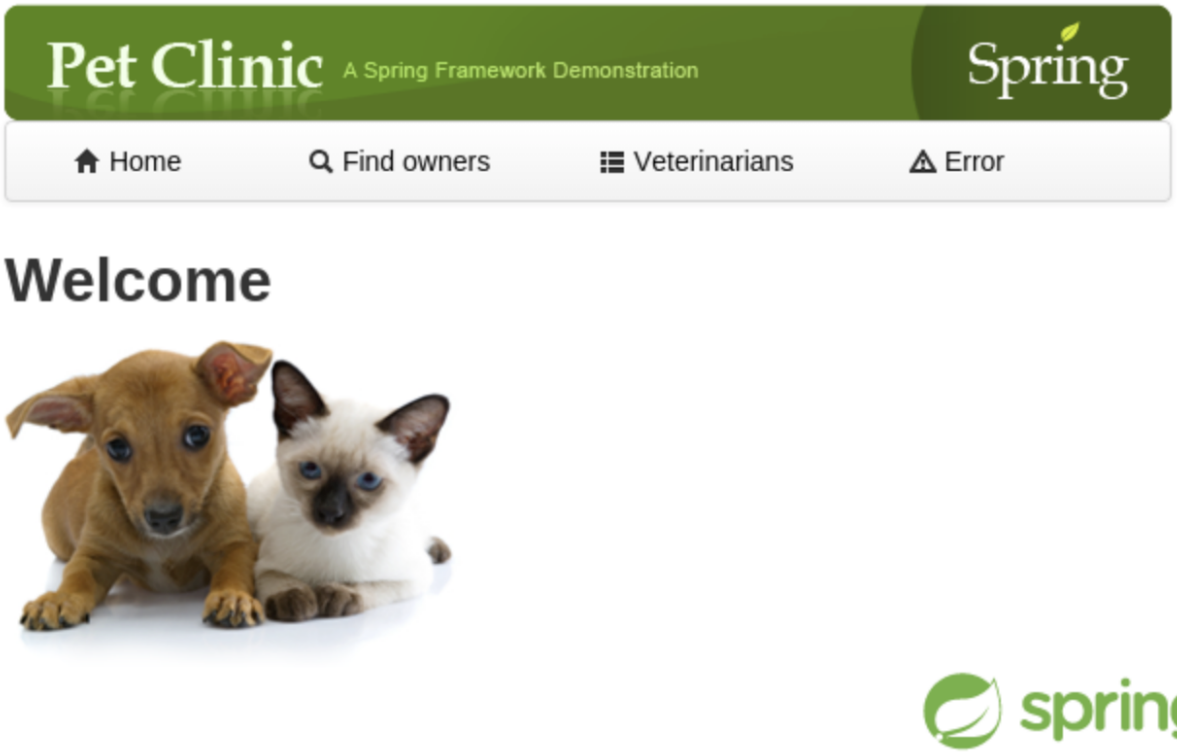
In sequential scripted automated tests/checks, we look at the expectation(s) using assert(s) statement(s), after we perform a set of well-known and predefined actions. Therefore, we can clearly say that the test scenario exercised by that test either passed or failed.

In MBT, especially in the case of State Transition Model-Based Testing, we start from a given vertex but then the path, that describes the sequence of edges and vertices visited, can be quite different each time the tool generates it. Besides, the stop condition is not composed by one or more well-known and fixed expectations; it's based on some more graph/model related criteria.

When we "execute the model", it will walk the path (i.e. go over from vertex to vertex through a given edge) and performing checks in the vertices. If those checks are successful until the stop condition(s) is achieved, we can say that it was successful; otherwise, the model is not a good representation of the system as it is we can say that it "failed."

Example

In this tutorial, we'll use an example provided by the GraphWalker community (please check GraphWalker wiki page describing it) which targets the well-known PetClinic sample site.



Requirements

- Java 8
- PetClinic sample application (requires Java 8 as it is)
    - ```
      git clone https://github.com/SpringSource/spring-petclinic.git
      cd spring-petclinic
      git reset --hard 482eeb1c217789b5d772f5c15c3ab7aa89caf279
      mvn tomcat7:run
      ```
- GraphWalker
- GraphWalker Studio

How can we test the PetClinic website using MBT technique?

Well, one approach could be to model the interactions between different pages. Ultimately they represent certain features that the site provides and that are connected with one another.

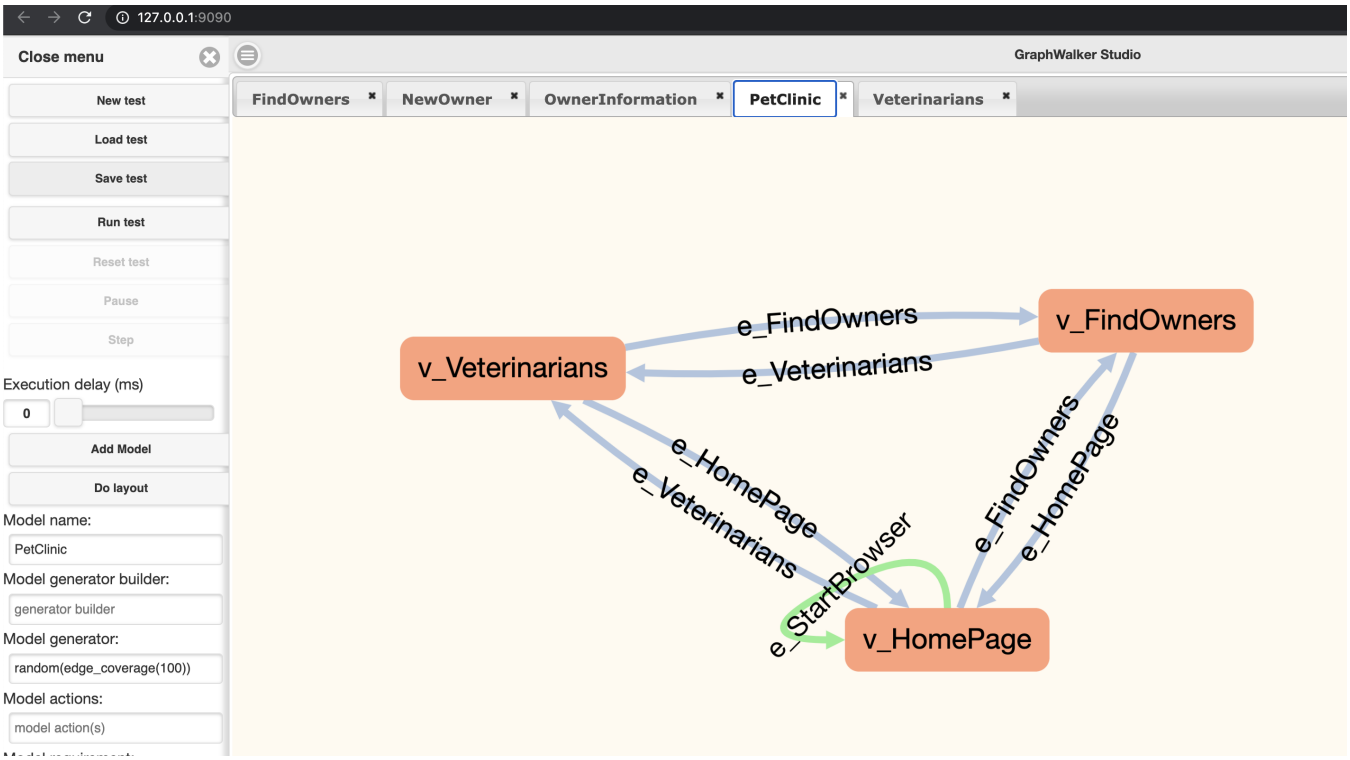
In this example, we'll be using these:

- **PetClinic**: main model of the PetClinic store, that relates several models provided by different sections in the site
- **FindOwners**: model around the feature of finding owners
- **Veterinarians**: model around the feature of listing veterinarians
- **OwnerInformation**: model around the ability of showing information/details of a owner
- **NewOwner**: model around the feature of creating a new owner

> **Please note**
>
> Remember that you could model it completely differently; modeling represents a perspective.

Models can be built using GraphWalker Studio. We can use it to load previously saved model(s) like the ones in PetClinic.json. In this case, the JSON file contains several models; we could also have one JSON file per model.

The following picture shows the overall PetClinic model, that interacts with other models.
      ```



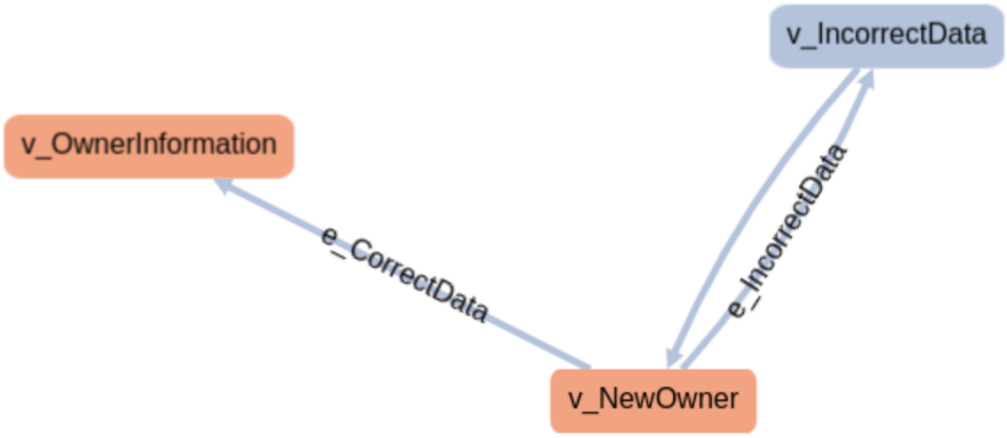
GraphWalker Studio allow us to run the model in offline, i.e. without executing the underlying test automation code, so we can validate it.

Let's pick the NewOwner model as an example, which is quite simple.

"v_NewOwner" represents, accordingly to what we've defined for our model, being on the "New Owner" page.

If we fill correct data (i.e. using the edge "e_CorrectData"), we'll be redirected to a page showing the owner information.

Otherwise, if we fill incorrect data (i.e. using the edge "e_IncorrectData") an error will be shown and the user keeps on the "New Owner" page.



Please note

Usually, to implement the automation code we would create a Maven project from scratch, copy the model file(s), and generate a skeleton of the sources for our model.

To do so, we would perform something such as:

```
# generate a Maven project prepared for GraphWalker
mvn archetype:generate -B -DarchetypeGroupId=org.graphwalker -DarchetypeArtifactId=graphwalker-maven-archetype -DgroupId=com.company -DartifactId=myProject

# store the JSON of the model(s) in src/main/resources/
...

# generate a skeleton of an implementable interface
mvn graphwalker:generate-sources
```

The Java class that implements the edges and vertices of this model is defined in the class NewOwnerTest. Actions performed in the edges are quite simple. Assertions are also simple as they're only focused on the state/vertex they are at.

class implementing the model "NewOwner"

```
package com.company.modelimplementations;

import com.company.NewOwner;
import com.github.javafaker.Faker;
import org.graphwalker.core.machine.ExecutionContext;
import org.graphwalker.java.annotation.GraphWalker;
import org.openqa.selenium.By;

import static com.codeborne.selenide.Condition.text;
import static com.codeborne.selenide.Condition.visible;
import static com.codeborne.selenide.Selenide.$;
import static com.codeborne.selenide.Selenide.$x;

/**
 * Implements the model (and interface) NewOwnerSharedState
 * The default path generator is Random Path.
 * Stop condition is 100% coverage of all edges.
 */
@GraphWalker(value = "random(edge_coverage(100))")
public class NewOwnerTest extends ExecutionContext implements NewOwner {

    @Override
    public void v_OwnerInformation() {
        $(By.tagName("h2")).shouldHave(text("Owner Information"));
        $x("/html/body/div/table[last()]/tbody/tr/td[2]/img").shouldBe(visible);
    }

    @Override
    public void e_CorrectData() {
        fillOwnerData();
        $(By.id("telephone")).sendKeys(String.valueOf(new Faker().number().digits(10)));
        $("button[type=\"submit\"]").click();
    }

    @Override
    public void e_IncorrectData() {
        fillOwnerData();
        $(By.id("telephone")).sendKeys(String.valueOf(new Faker().number().digits(20)));
        $("button[type=\"submit\"]").click();
    }

    @Override
    public void v_IncorrectData() {
        $(By.cssSelector("div.control-group.error > div.controls > span.help-inline"))
                .shouldHave(text("numeric value out of bounds (<10 digits>.<0 digits> expected)"));
    }

    @Override
    public void v_NewOwner() {
        $(By.tagName("h2")).shouldHave(text("New Owner"));
        $x("/html/body/table/tbody/tr/td[2]/img").shouldBe(visible);
    }

    private void fillOwnerData() {
        $(By.id("firstName")).clear();
        $(By.id("firstName")).sendKeys(new Faker().name().firstName());

        $(By.id("lastName")).clear();
        $(By.id("lastName")).sendKeys(new Faker().name().lastName());

        $(By.id("address")).clear();
        $(By.id("address")).sendKeys(new Faker().address().fullAddress());

        $(By.id("city")).clear();
        $(By.id("city")).sendKeys(new Faker().address().city());

        $(By.id("telephone")).clear();
    }
}
```

In the previous example, we can see that the class NewOwnerTest extends ExecutionContext; this ties the model with the path generator and provides a context for tracking the internal state and history of the model.

The **@GraphWalker** annotation is used to specify the path generator and stop conditions. This is used for *online* path generation during test execution.

If follows this syntax:

```
@GraphWalker(value = "generator(stop_conditions)", start = "start_element", groups = { "group" }  )
```

such as:

```
@GraphWalker(value = "random(reached_vertex(v_ShoppingCart))", start = "e_StartBrowser", groups =  { "default" } )
```

Please note

Tests using the model can also be created and executed programmatically similar to other tests, using JUnit or other testing framework. More info here and here.

The flow would be something like:

1. create a TestBuilder object
2. create a Context object
3. add the Context to the TestBuilder
4. execute it, using .execute()
5. optionally, look at the Result object returned to see if it has errors, using .hasErrors()

example of some Tests implementing using JUnit

```
public class SimpleTest extends ExecutionContext implements Login {
    public final static Path MODEL_PATH = Paths.get("org/myorg/testautomation/Login.json");
...
    @Test
    public void runSmokeTest() {
        new TestBuilder()
                .addContext(new SimpleTest().setNextElement(new Edge().setName("e_Init").build()),
                        MODEL_PATH,
                        new AStarPath(new ReachedVertex("v_Browse")))
                .execute();
    }

    @Test
    public void runFunctionalTest1() {
        new TestBuilder()
                .addContext(new SimpleTest().setNextElement(new Edge().setName("e_Init").build()),
                        MODEL_PATH,
                        new RandomPath(new EdgeCoverage(100)))
                .execute();
    }

    @Test
    public void runFunctionalTest2() {
        TestBuilder builder = new TestBuilder()
                .addContext(new SimpleTest().setNextElement(new Edge().setName("e_Init").build()),
                        MODEL_PATH,
                        new RandomPath(new EdgeCoverage(100)));
        Result result = builder.execute(true);
        Assert.assertFalse(result.hasErrors());
    }

    @Test
    public void runStabilityTest() {
        new TestBuilder()
                .addContext(new SimpleTest().setNextElement(new Edge().setName("e_Init").build()),
                        MODEL_PATH,
                        new RandomPath(new TimeDuration(30, TimeUnit.SECONDS)))
                .execute();
    }
}
```

In this case, we could execute the tests using Maven. We would then use the JUnit XML report produced by JUnit itself.

```
mvn test
```

To run the tests online with GraphWalker we can use Maven, since there is a specific plugin for assisting with this. This will produce a single JUnit XML report stored in the `target/graphwalker-reports/` directory.

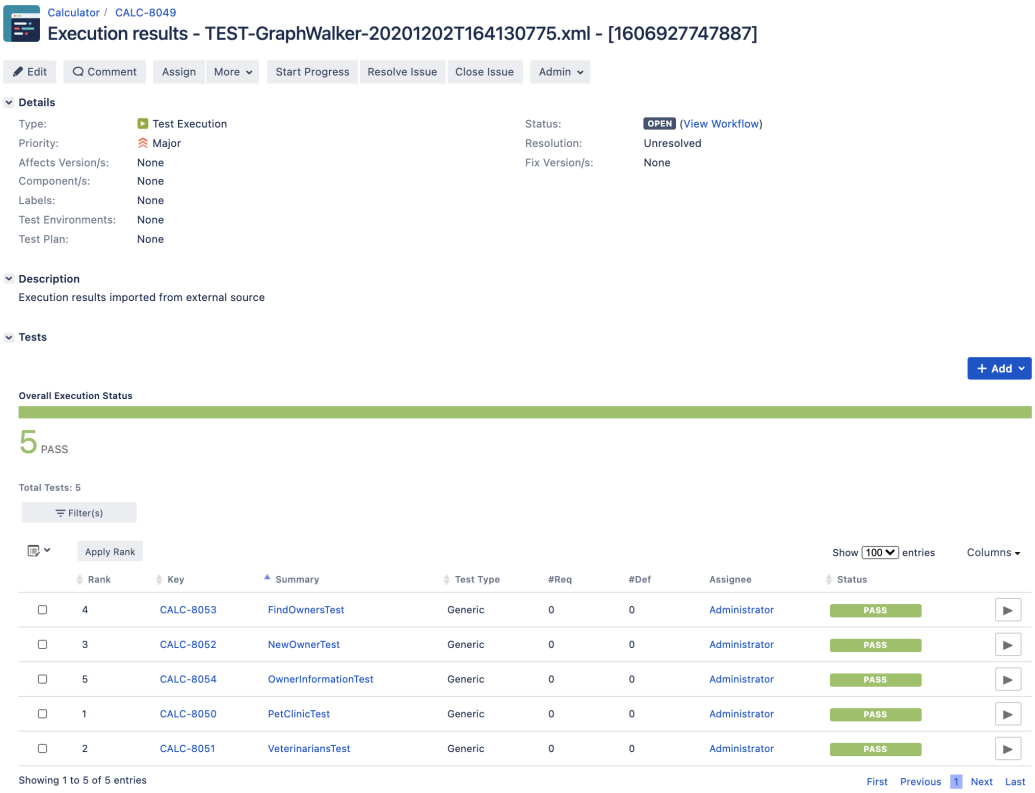
example of a Bash script to run the tests

```
rm -f target/graphwalker-reports/*.xml
mvn graphwalker:test
```

After successfully running the tests and generating the JUnit XML report, it can be imported to Xray (either by the REST API or through the **Import Execution Results** action within the Test Execution, or even by using a CI tool of your choice).

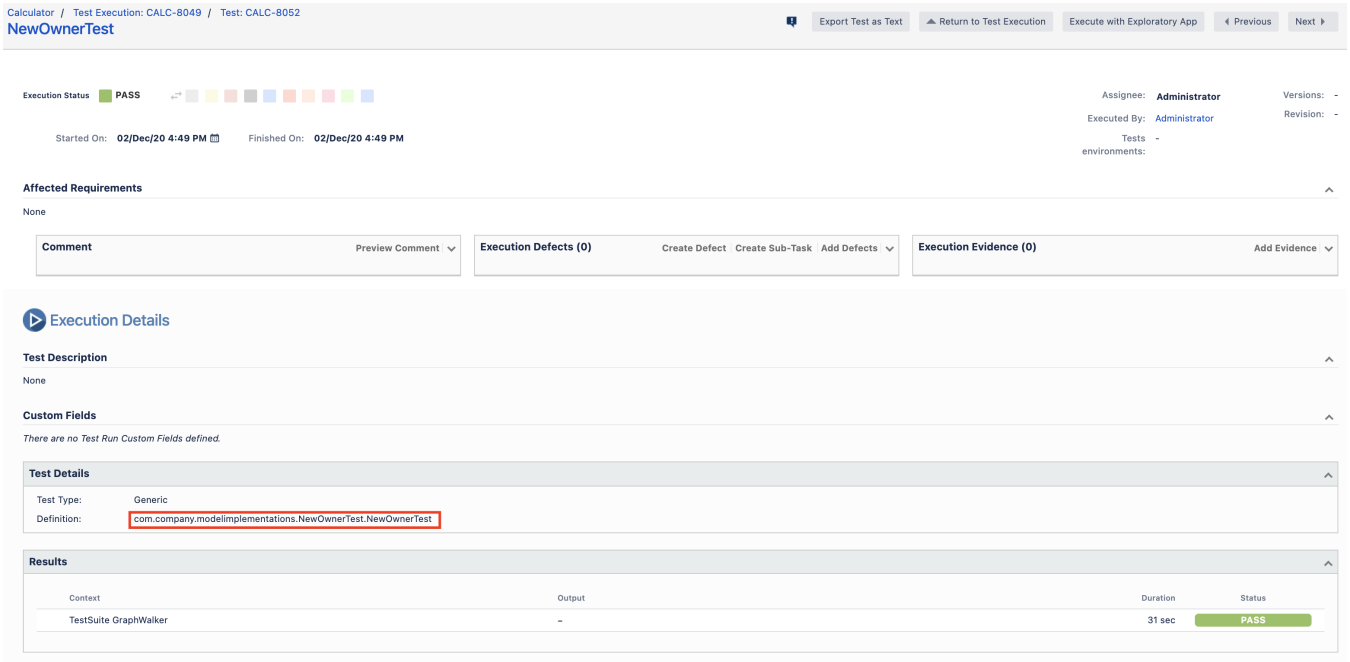
example of a Bash script to import the results

```
REPORT_FILE=$(ls target/graphwalker-reports/TEST-GraphWalker-*.xml | sort | tail -n 1)
curl -H "Content-Type: multipart/form-data" -u admin:admin -F "file=@$REPORT_FILE" http://jiraserver.example
/rest/raven/1.0/import/execution/junit?projectKey=CALC
```



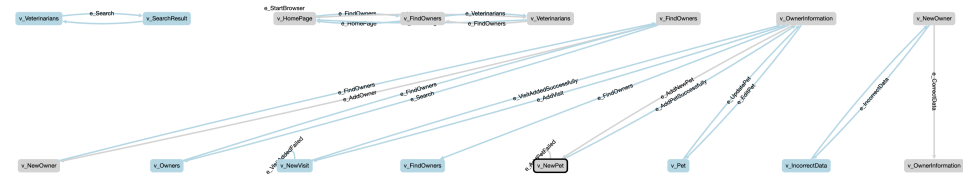
Each model is mapped to JUnit's testcase element which in turn is mapped to a Generic Test in Jira, and the **Generic Test Definition** field contains the name of the package and the class that implements the model related methods for edges and vertices. The summary of each Test issue is filled out with the name of the class.

The Execution Details page also shows information about the Test Suite, which will be just "GraphWalker."

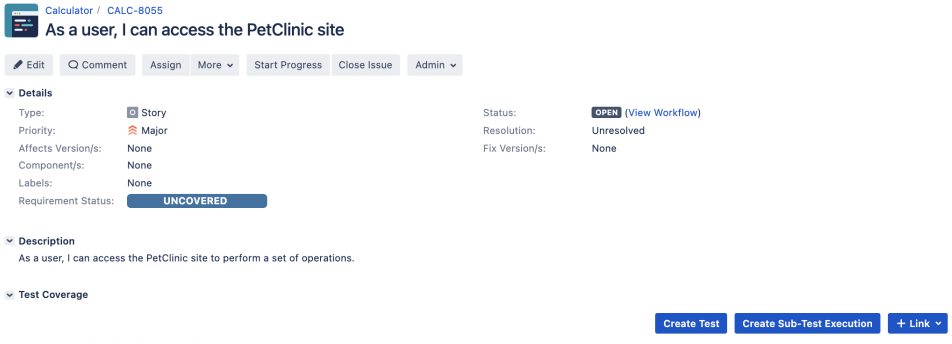


Tips

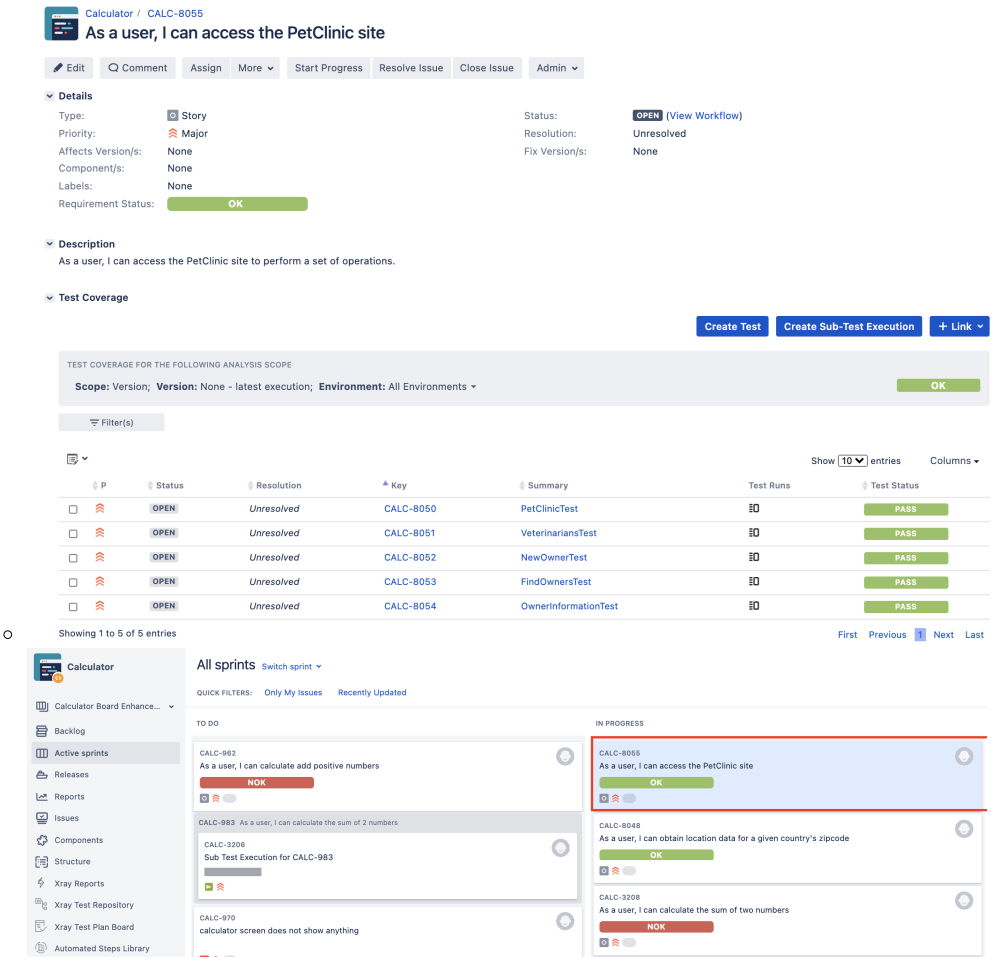
- Use MBT not to replace existing test scripts but in cases where you need to provide greater coverage
- Discuss the model(s) with the team and the ones that can be most useful for your use case
- You can control the seed of the random generator used by GraphWalker, so you can easily reproduce bugs (i.e. by reproducing the generated path)
- You can use GraphWalker Player to see the graph being walked in real-time. You can use a sample HTML file that contains the code to connect to a WebSocket server that you need to instantiate in the runner side (example) .
    - Example:
        - open the file index.html in your browser, using an URL such as "file:///Users/you/index.html?wsURI=localhost:8887?wsURI=localhost:8887"
        - execute GraphWalker, using the custom runner
            - `mvn exec:java -Dexec.mainClass="com.company.runners.WebSocketApplication"`



- Multiple runs of your tests can be grouped and consolidate in a Test Plan, so you can have an updated overview of their current state
- After importing the results, you can link the corresponding Test issues with an existing requirement or user story and thus track coverage directly on the respective issue, or even on an Agile board



    - No Tests were found testing the requirement.



References

- GraphWalker
- GraphWalker documentation pages
- GraphWalker model+code for testing the PetClinic site
- Actions and Guards (from AltkWalker's documentation)
- GraphWalker CLI
- GraphWalker Player
- GraphWalker plugin for Eclipse (GW4E)
- GraphWalker and GW4E in a nutshell
- Article on MBT