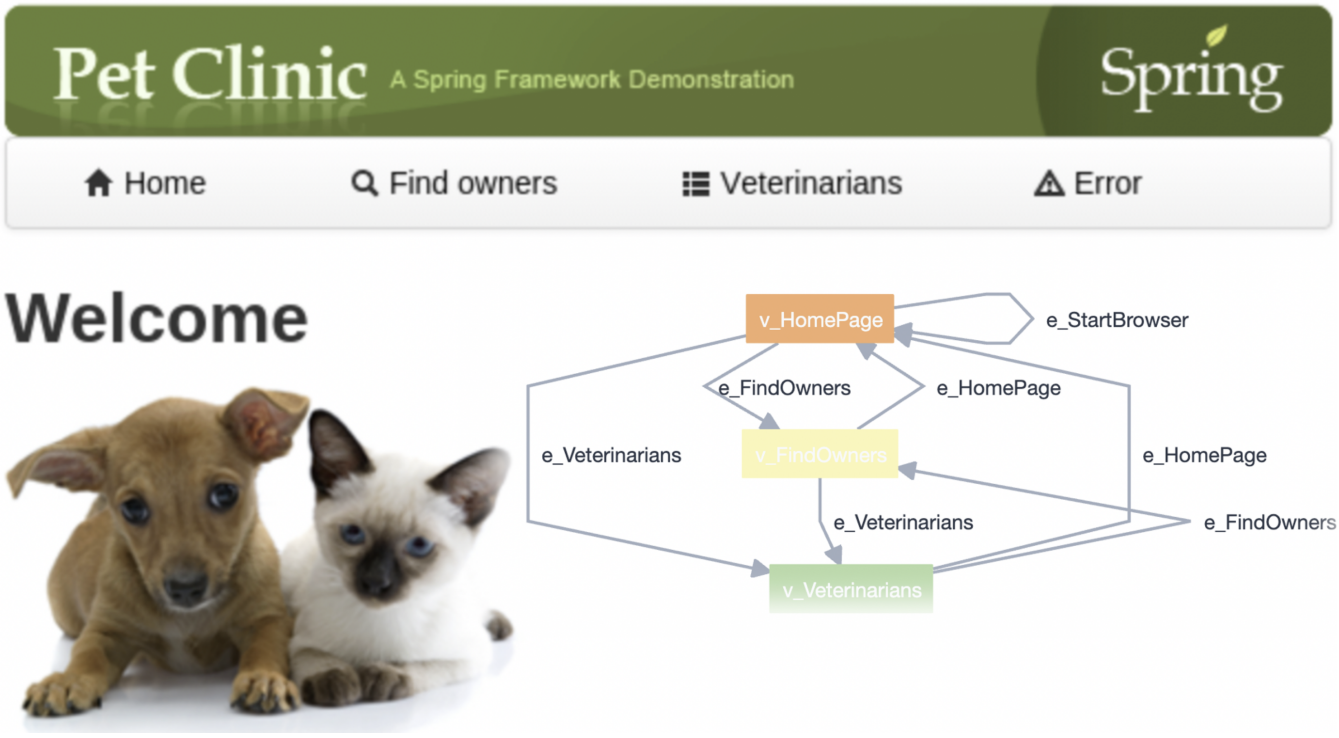
Model-Based Testing using AltWalker and Python

Overview

AltWalker is a test execution tool targeted for Model-Based Testing which interacts closely with GraphWalker.

GraphWalker 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.

With AltWalker, automation code related to our model can be implemented in Python, C#, or other. In this approach, GraphWalker is only responsible for generating the path through the models.



Let's clarify 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 AltWalker & GraphWalker (more info here, 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. AltWalker will then "walk" the model (i.e. perform a set of "steps"/edges) using a generated path from GraphWalker. While doing so, it looks at JavaScript guards to check is 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 either use a visual tool (AltWalker's Model-Editor, or GraphWalker Studio) and export it to a JSON file, or an IDE instead (e.g. VSCode with a specific extension).

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 best 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.

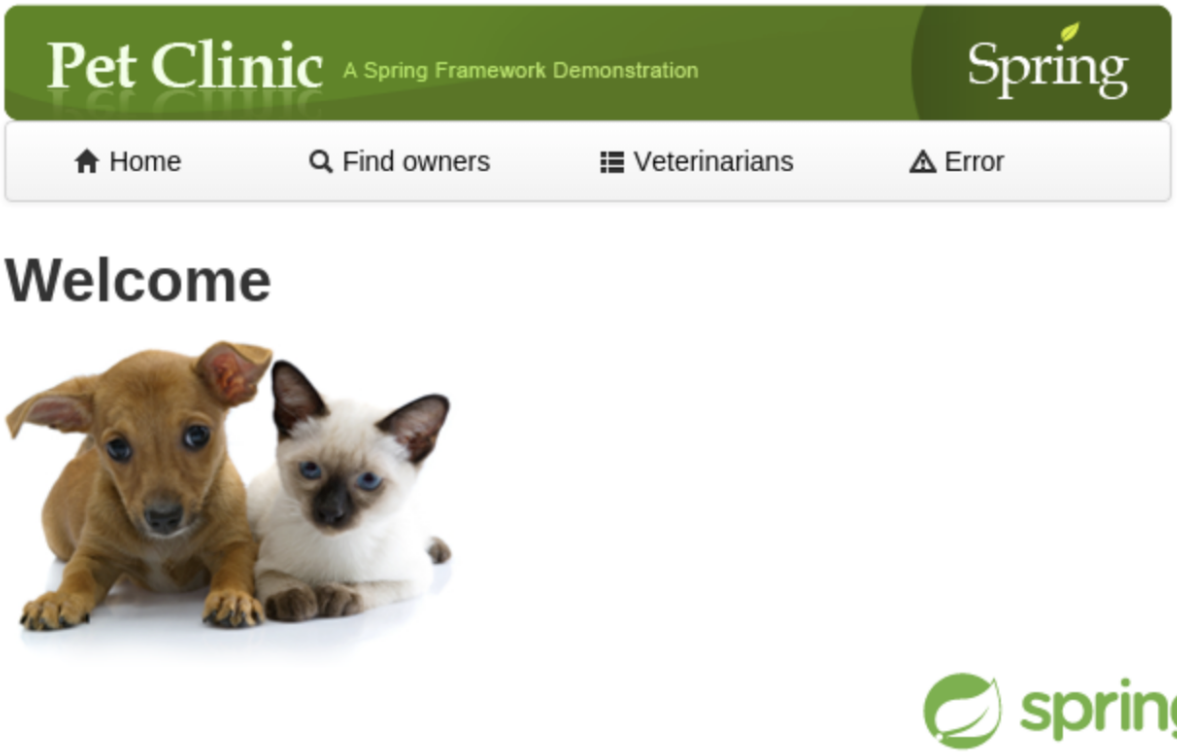
In MBT, especially in the case of State Transition Model-Based Testing, we start from a given vertex but then the path, that describes the sequence of edges and vertices visited, can be quite different each time the tool generates it. The stop condition is not composed by one or more well-known and fixed 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 perform 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 and we can say that it "failed."

Example

This tutorial is based on an example provided by the GraphWalker community (please check GraphWalker wiki page describing it) which targets the well-known PetClinic sample site.

This example has been ported from GraphWalker+Java to AltWalker+Python and the full source-code is available here.



Requirements

- Target SUT (PetClininc sample application):
  - Java 8
  - source-code
    - ```
      git clone https://github.com/SpringSource/spring-petclinic.git
      cd spring-petclinic
      git reset --hard 482eeb1c217789b5d772f5c15c3ab7aa89caf279
      mvn tomcat7:run
      ```
- Test code (source-code and additional details here)
  - GraphWalker 4.2.0
  - AltWalker 0.2.7
  - Altom's Model-Editor or GraphWalker Studio

How can we test the PetClinic 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.

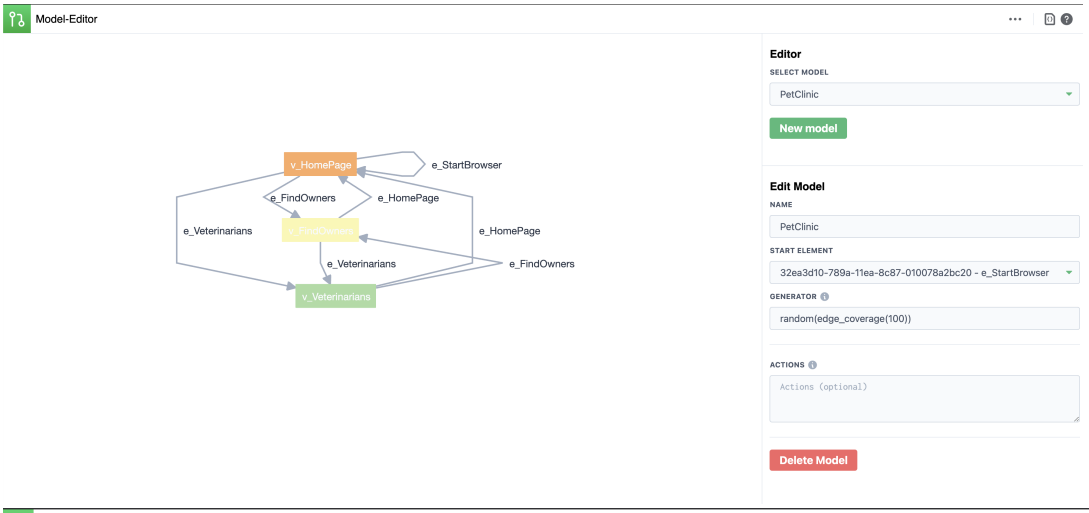
In this example, we'll be using these:

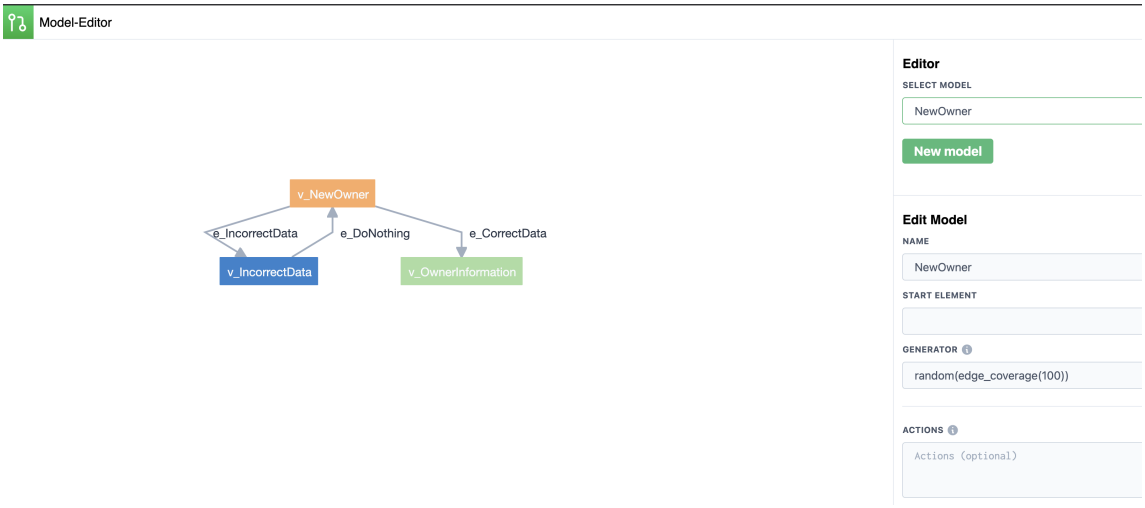
- **PetClinic**: main model of the PetClinic store, that relates several models provided by different sections in the site
- **FindOwners**: model around the feature of finding owners
- **Veterinarians**: model around the feature of listing veterinarians
- **OwnerInformation**: model around the ability 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.

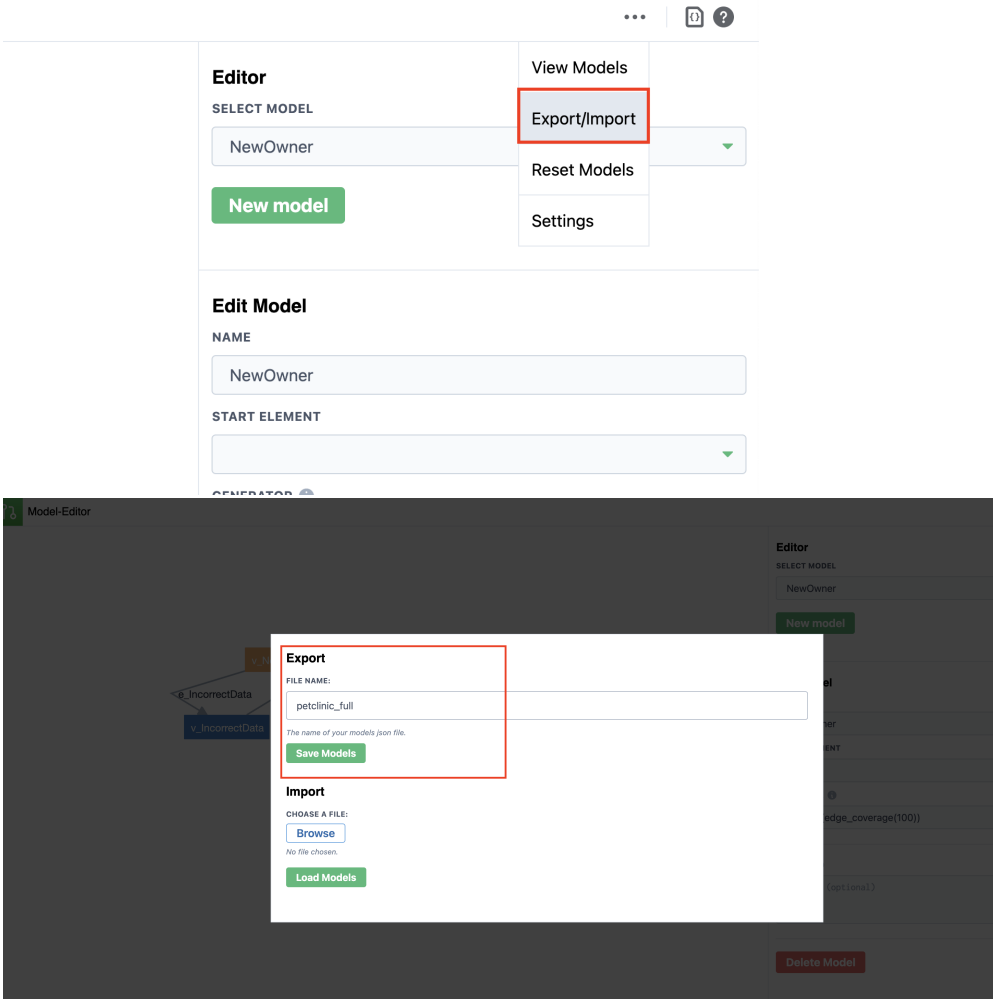
As mentioned earlier, models can be built using AltWalker's Model-Editor (or GraphWalker Studio) or directly in the IDE (for VSCode there's a useful extension to preview it). In the visual editors, namely in AltWalker's Model-Editor, we can use it to load previously saved model(s) like the ones in petclinic_full.json. 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, and also the NewOwner model.





If we use the visual editors to build the model, then we need to export it to one (or more) JSON file(s).



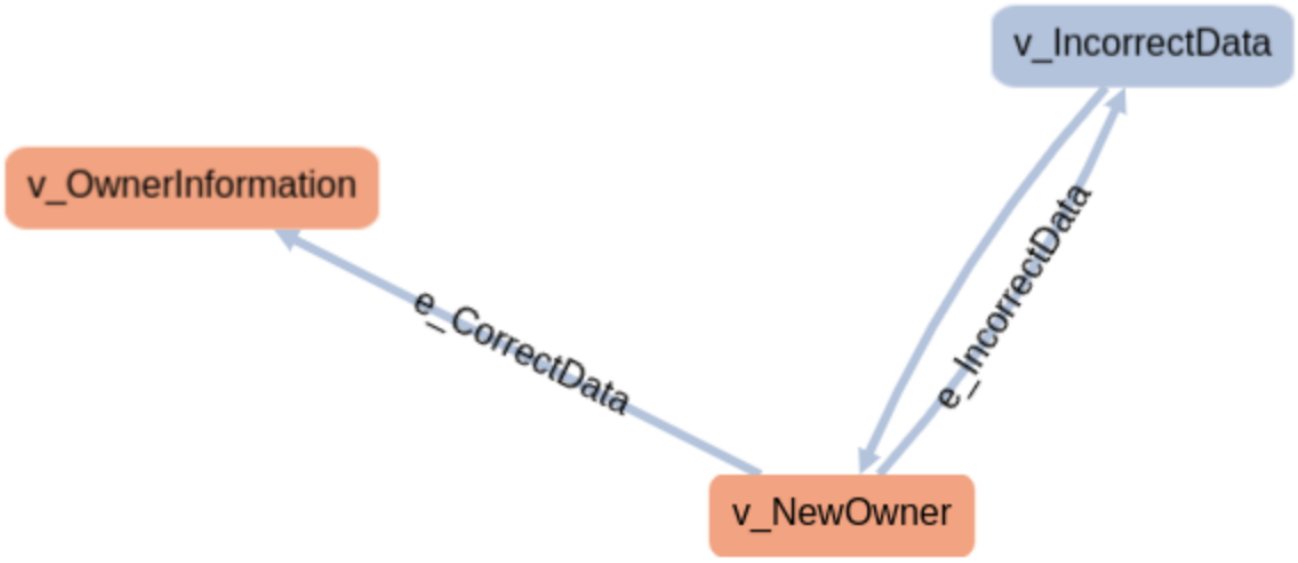
Note: if you use GraphWalker Studio instead, it allows you to run the model in offline, i.e. without executing the underlying test automation code, so we can validate it.

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

As detailed in AltWalker's documentation, if we start from scratch (i.e. without a model), we can initialize a project for our automation code using something like:

```
$ altwalker init -l python test-project
```

When we have the model, we can generate the test package containing a skeleton for the underlying test code.

```
$ altwalker generate -l python path/for/test-project/ -m path/to/models.json
```

If we do have a model, then we can pass it to the initialization command:

```
$ altwalker init -l python test-project -m path/to/model-name.json
```

During implementation, we can check our model for issues/inconsistencies, just from a modeling perspective:

```
$ altwalker check -m path/to/model-name.json "random(vertex_coverage(100))"
```

We can also check verify if the test package contains the implementation of the code related to the vertices and edges.

```
$ altwalker verify -m path/to/model-name.json tests
```

Check the full syntax of AltWalker's CLI (i.e. "altwalker") for additional details.

The main test package is stored in tests/test.py. The implementation follows the Page Objects Model using pypom package and each page is stored in a proper class under a specific pages directory.

Besides, faker is also used to generate test data that will be used by the model (e.g. whenever filling data on the edges).

Actions performed in the edges are quite simple. Assertions are also simple as they're only focused on the state/vertex they are at.

tests/test.py (main code with the tests)

```
import unittest

from selenium import webdriver
from selenium.webdriver.firefox.options import Options

from tests.pages.base import BasePage
from tests.pages.home import HomePage
from tests.pages.find_owners import FindOwnersPage
from tests.pages.owners import OwnersPage
from tests.pages.new_owner import NewOwnerPage
from tests.pages.veterinarians import VeterinariansPage
from tests.pages.owner_information import OwnerInformationPage

import sys
import pdb
from faker import Faker

debugger = pdb.Pdb(skip=['altwalker.*'], stdout=sys.stdout)
fake = Faker()

HEADLESS = False
BASE_URL = "http://localhost:9966/petclinic"

driver = None


def setUpRun():
    """Setup the webdriver."""

    global driver

    options = Options()
    if HEADLESS:
        options.add_argument('-headless')

    print("Create a new Firefox session")
    driver = webdriver.Firefox(options=options)

    print("Set implicitly wait")
    driver.implicitly_wait(15)
    print("Window size: {width}x{height}".format(**driver.get_window_size()))

def tearDownRun():
    """Close the webdriver."""

    global driver

    print("Close the Firefox session")
    driver.quit()

class BaseModel(unittest.TestCase):
        """Contains common methods for all models."""

        def setUpModel(self):
                global driver
                print("Set up for: {}".format(type(self).__name__))
                self.driver = driver

        def v_HomePage(self):
```

```
                page = HomePage(self.driver)
                self.assertEqual(page.heading_text, "Welcome", "Welcome heading should be present")
                self.assertTrue(page.is_footer_present, "footer should be present")

        def v_FindOwners(self):
                page = FindOwnersPage(self.driver)
                self.assertEqual("Find Owners",page.heading_text, "Find Owners heading should be present")
                self.assertTrue(page.is_footer_present, "footer should be present")

        def v_NewOwner(self):
                page = NewOwnerPage(self.driver)
                self.assertEqual( "New Owner",page.heading_text, "New Owner heading should be present")
                #$x("/html/body/table/tbody/tr/td[2]/img").shouldBe(visible);
                self.assertTrue(page.is_footer_present, "footer should be present")

        def v_Owners(self):
                page = OwnersPage(self.driver)
                self.assertEqual("Owners",page.heading_text, "Owners heading should be present")
                self.assertGreater(page.total_owners_in_list, 9, "Owners in listing >= 10")

        def v_Veterinarians(self):
                page = VeterinariansPage(self.driver)
                self.assertEqual(page.heading_text,"Veterinarians", "Veterinarians heading should be present")
                self.assertTrue(page.is_footer_present, "footer should be present")

        def v_OwnerInformation(self, data):
                page = OwnerInformationPage(self.driver)
                self.assertEqual(page.heading_text, "Owner Information", "Owner Information heading should be
present")
                data["numOfPets"] = page.number_of_pets
                print(f"numOfPets: {page.number_of_pets}")
                self.assertTrue(page.is_footer_present, "footer should be present")

        def e_DoNothing(self, data):
                #debugger.set_trace()
                pass

        def e_FindOwners(self):
                page = BasePage(self.driver)
                page.click_find_owners()


class PetClinic(BaseModel):
        def e_StartBrowser(self):
                page = HomePage(self.driver, BASE_URL)
                page.open()

        def e_HomePage(self):
                page = HomePage(self.driver)
                page.click_home()

        def e_Veterinarians(self):
                page = HomePage(self.driver)
                page.click_veterinarians()

        def e_FindOwners(self):
                page = HomePage(self.driver)
                page.click_find_owners()

class FindOwners(BaseModel):

        def e_AddOwner(self):
                page = FindOwnersPage(self.driver)
                page.click_add_owner()

        def e_Search(self):
                page = FindOwnersPage(self.driver)
                page.click_submit()


class OwnerInformation(BaseModel):
```

```
        def e_UpdatePet(self):
                page = OwnerInformationPage(self.driver)
                page.click_submit()

        def e_AddPetSuccessfully(self):
                page = OwnerInformationPage(self.driver)
                page.fillout_pet(fake.name(),fake.past_date().strftime("%Y/%m/%d"), "dog")
                page.click_submit()

        def e_AddPetFailed(self):
                page = OwnerInformationPage(self.driver)
                page.fillout_pet("",fake.past_date().strftime("%Y/%m/%d"), "dog")
                page.click_submit()

        def e_AddNewPet(self):
                page = OwnerInformationPage(self.driver)
                page.click_add_new_pet()

        def e_EditPet(self):
                page = OwnerInformationPage(self.driver)
                page.click_edit_pet()

        def e_AddVisit(self):
                page = OwnerInformationPage(self.driver)
                page.click_add_visit()

        def v_NewPet(self):
                page = OwnerInformationPage(self.driver)
                self.assertEqual(page.heading_text, "New Pet", "New Pet heading should be present")
                self.assertTrue(page.is_footer_present, "footer should be present")

        def v_NewVisit(self):
                page = OwnerInformationPage(self.driver)
                self.assertEqual(page.heading_text, "New Visit", "New Visit heading should be present")
                self.assertTrue(page.is_visit_visible, "visit should be present")

        def e_VisitAddedSuccessfully(self):
                page = OwnerInformationPage(self.driver)
                page.clear_description()
                page.set_description(fake.name())
                page.click_submit()

        def e_VisitAddedFailed(self):
                page = OwnerInformationPage(self.driver)
                page.clear_description()
                page.click_submit()

        def v_Pet(self):
                page = OwnerInformationPage(self.driver)
                self.assertEqual(page.heading_text, "Pet", "Pet heading should be present")

class Veterinarians(BaseModel):
        def e_Search(self):
                page = VeterinariansPage(self.driver)
                page.search_for("helen")

        def v_SearchResult(self):
                page = VeterinariansPage(self.driver)
                self.assertTrue(page.is_text_present_in_vets_table, "Helen Leary")
                self.assertTrue(page.is_footer_present, "footer should be present")

        def v_Veterinarians(self):
                page = VeterinariansPage(self.driver)
                self.assertEqual(page.heading_text,"Veterinarians", "Veterinarians heading should be present")
                self.assertGreater(page.number_of_vets_in_table, 0, "At least one Veterinarian should be listed
in table")

class NewOwner(BaseModel):

        def e_CorrectData(self):
                page = NewOwnerPage(self.driver)
```

```
                page.fill_owner_data(first_name=fake.first_name(), last_name=fake.last_name(), address=fake.
address(), city=fake.city(), telephone=fake.pystr_format('##########'))
                #page.fill_telephone(fake.pystr_format('##########'))
                page.click_submit()

        def e_IncorrectData(self):
                page = NewOwnerPage(self.driver)
                page.fill_owner_data()
                #page.fill_telephone("12345678901234567890")
                page.fill_telephone(fake.pystr_format('####################'))
                page.click_submit()

        def v_IncorrectData(self):
                page = NewOwnerPage(self.driver)
                self.assertTrue(page.error_message, "numeric value out of bounds (<10 digits>.<0 digits>
expected")
```

In the previous code, we can see that each model is a class. Each one of those classes must contain methods corresponding to the related edges and vertices; methods should be named in the same way as the names assigned for the edges and for the vertices in the model.

To run the tests using a random path generator and stopping upon 100% of vertex coverage, we can use AltWalker CLI tool such as:

example of a Bash script to run the tests

```
altwalker online tests -m models/petclinic_full.json "random(vertex_coverage(100))"
```

However, that would only produce some debug output to the console.

If we aim to integrate this in CI/CD, or even have visibility of it in a test management tool such as Xray, we need to generate a JUnit XML report.

However, AltWalker (as of v0.2.7) does not yet provide a built-in JUnit reporter.

Luckily, we can implement our own code to run AltWalker as it provides an open API. This code is available in the script run_with_custom_junit_report.py, which can be found the repository the sample code of this tutorial.

example of Python code to run the tests with a custom reporter

```
from altwalker.planner import create_planner
from altwalker.executor import create_executor
from altwalker.walker import create_walker
from custom_junit_reporter import CustomJunitReporter
import sys
import pdb
import click

def _percentege_color(percentage):
    if percentage < 50:
        return "red"

    if percentage < 80:
        return "yellow"

    return "green"


def _style_percentage(percentege):
    return click.style("{}%".format(percentege), fg=_percentege_color(percentege))
```

```
def _style_fail(number):
    color = "red" if number > 0 else "green"

    return click.style(str(number), fg=color)


def _echo_stat(title, value, indent=2):
    title = " " * indent + title.ljust(30, ".")
    value = str(value).rjust(15, ".")

    click.echo(title + value)

def _echo_statistics(statistics):
    """Pretty-print statistics."""

    click.echo("Statistics:")
    click.echo()

    total_models = statistics["totalNumberOfModels"]
    completed_models = statistics["totalCompletedNumberOfModels"]
    model_coverage = _style_percentage(completed_models * 100 // total_models)

    _echo_stat("Model Coverage", model_coverage)
    _echo_stat("Number of Models", click.style(str(total_models), fg="white"))
    _echo_stat("Completed Models", click.style(str(completed_models), fg="white"))

    _echo_stat("Failed Models", _style_fail(statistics["totalFailedNumberOfModels"]))
    _echo_stat("Incomplete Models", _style_fail(statistics["totalIncompleteNumberOfModels"]))
    _echo_stat("Not Executed Models", _style_fail(statistics["totalNotExecutedNumberOfModels"]))
    click.echo()


debugger = pdb.Pdb(skip=['altwalker.*'], stdout=sys.stdout)
reporter = None

if __name__ == "__main__":
    try:
        planner = None
        executor = None
        statistics = {}
        models = [("models/petclinic_full.json","random(vertex_coverage(100))")]
        steps = None
        graphwalker_port = 5000
        start_element=None
        url="http://localhost:5000/"
        verbose=False
        unvisited=False
        blocked=False
        tests = "tests"
        executor_type = "python"
        planner = create_planner(models=models, steps=steps, port=graphwalker_port, start_element=start_element,
                                 verbose=True, unvisited=unvisited, blocked=blocked)
        executor = create_executor(tests, executor_type, url=url)
        reporter = CustomJunitReporter()

        walker = create_walker(planner, executor, reporter=reporter)
        walker.run()
        statistics = planner.get_statistics()
    finally:
        print(statistics)
        _echo_statistics(statistics)
        reporter.set_statistics(statistics)
        junit_report = reporter.to_xml_string()
        print(junit_report)
        with open('output.xml', 'w') as f:
            f.write(junit_report)
        with open('output_allinone.xml', 'w') as f:
            f.write(reporter.to_xml_string(generate_single_testcase=True, single_testcase_name="
PetClinicAllinOne"))

        #debugger.set_trace()
```

```
        if planner:
            planner.kill()

        if executor:
            executor.kill()
```

This code makes use of a custom reporter that can generate JUnit XML reports in two different ways:

1. mapping each model to a JUnit <testcase> element, which ultimately will be translated to a Test issue in Xray per each model
2. mapping the whole run to a single JUnit <testcase> element, considering the whole run as successful or not; in this case, it will be lead to a single Test issue in Xray

The previous runner's code above produces these two reports, so we can evaluate them.

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).

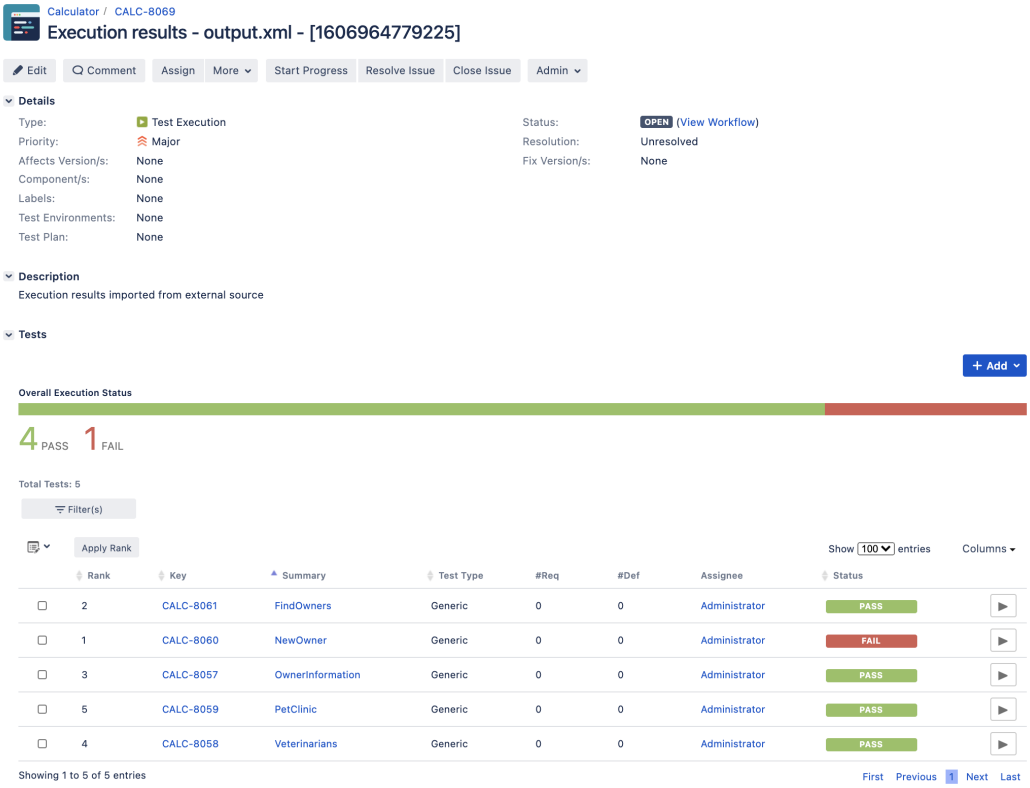
example of a Bash script to import the results

```
#!/bin/bash

# if you wish to map the whole run to single Test in Xray/Jira
#REPORT_FILE=output_allinone.xml

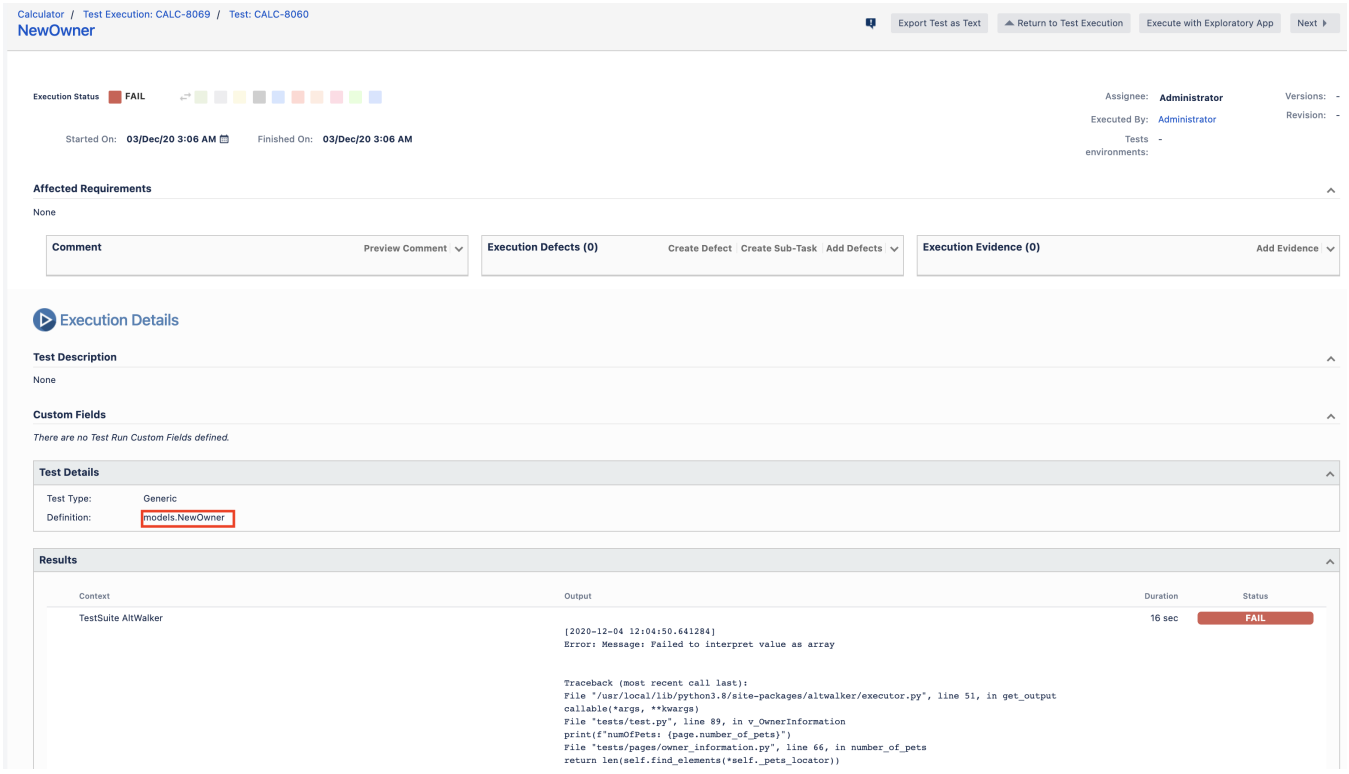
# if you wish to map each model as a separate Test in Xray/Jira
REPORT_FILE=output.xml

curl -H "Content-Type: multipart/form-data" -u admin:admin -F "file=@$REPORT_FILE" http://jiraserver.example.com
/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 unique identifier of our test; in this case it's "model.<name_of_model>". The summary of each Test issue has the name of the model.

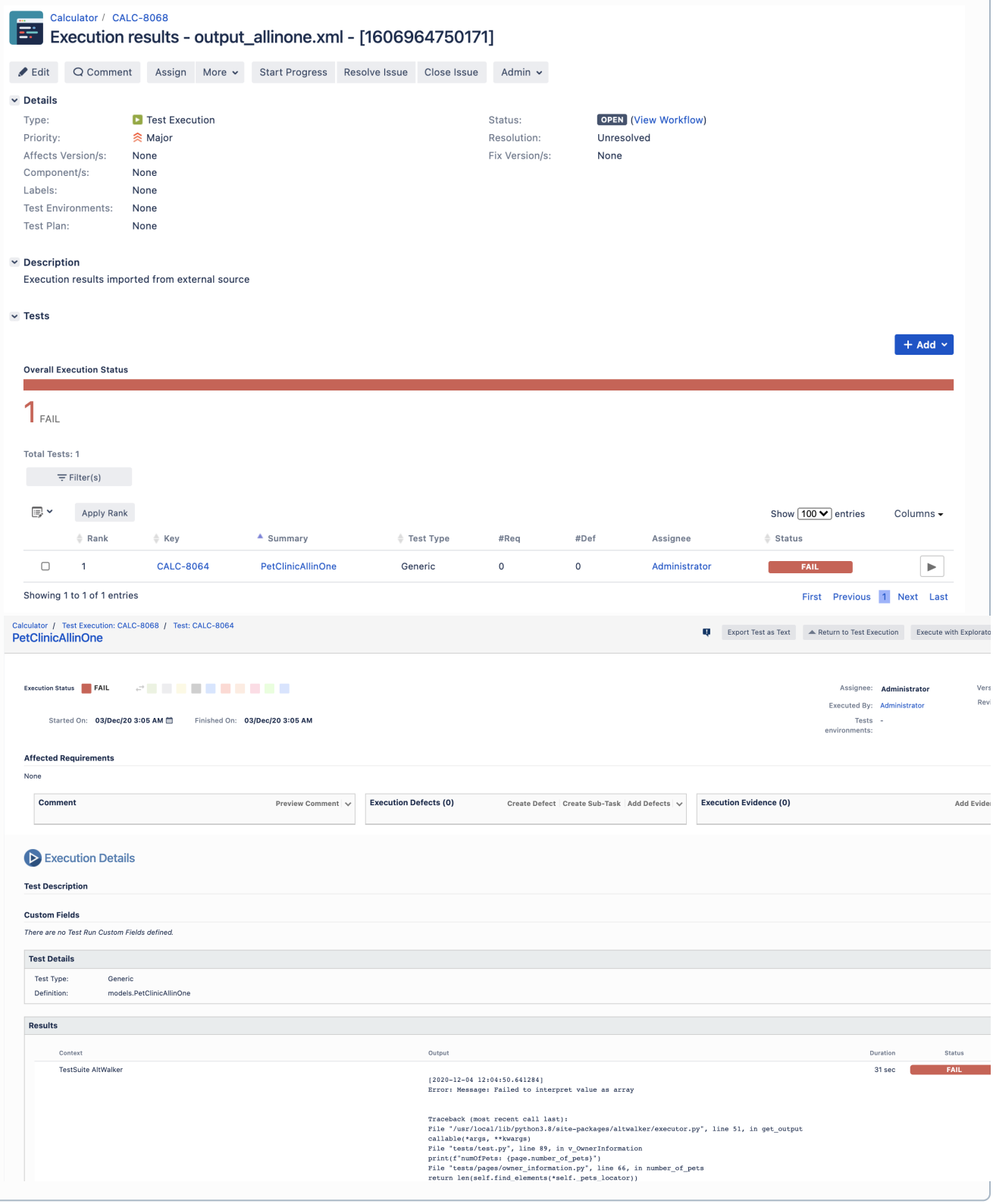
The Execution Details page also shows information about the Test Suite, which will be just "AltWalker".



Alternate JUnit XML generation (all-in-one/single testcase)

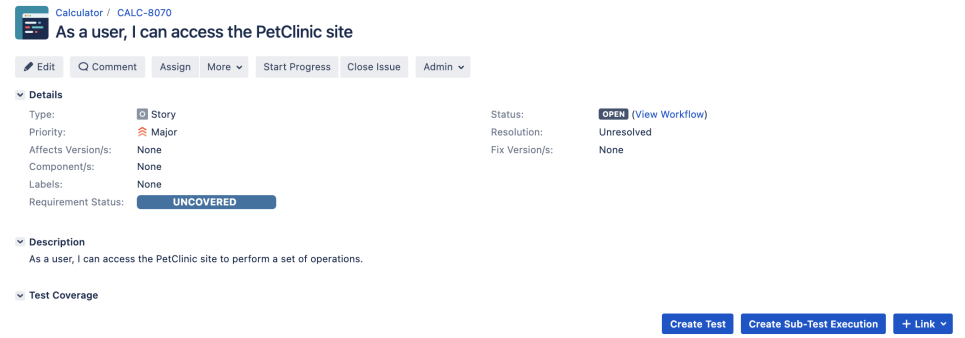
If we generate the JUnit XML report with a single <testcase> element for the whole run of our model, we would have just one Test created in Xray. It would be globally passed/failed.

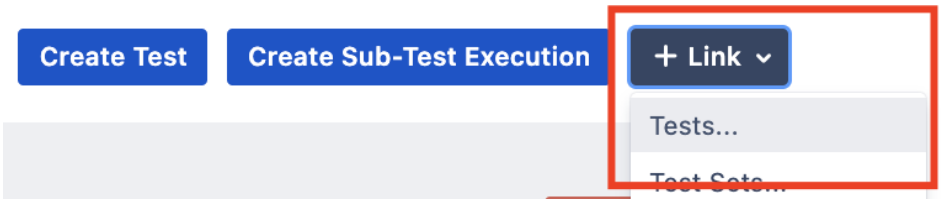
Our complete model is abstracted to a Test issue having a Generic Test Definition (i.e. its unique identifier) as something as "models.<customizable_in_the_reporter>".

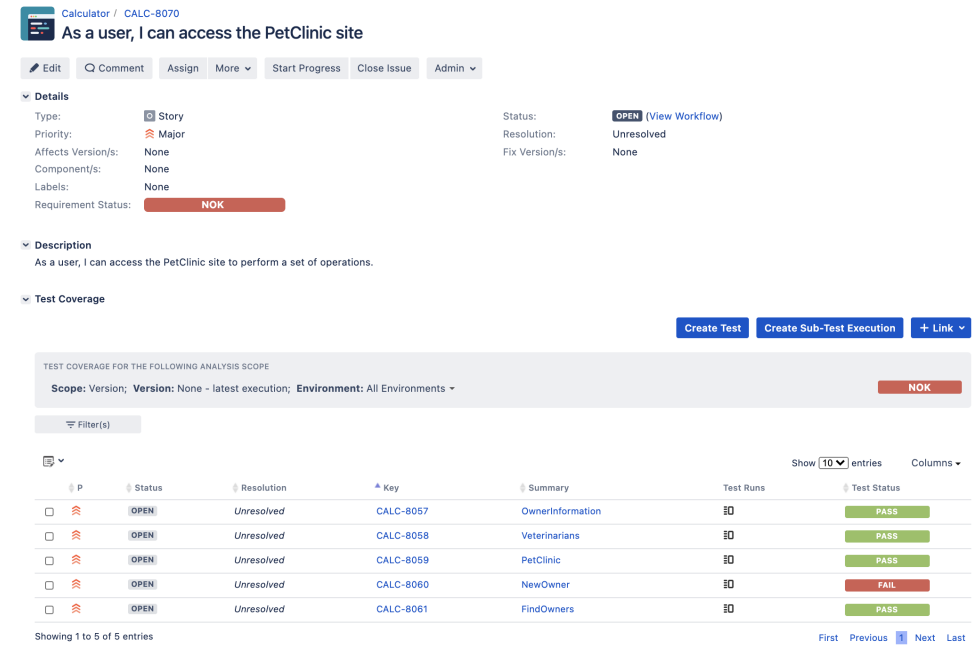


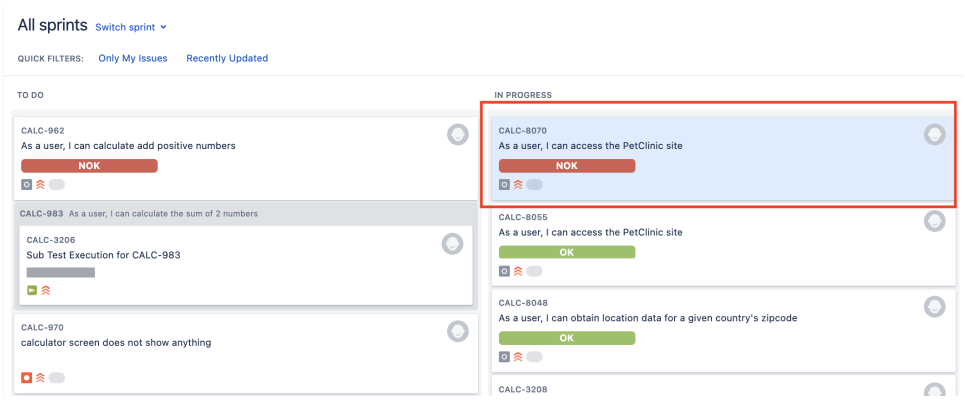
Tips

- 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 valuable for your use case
- Multiple runs of your tests can be grouped and consolidated 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 truck coverage directly on the respective issue, or even on an Agile board









References

- AltWalker
- Visual model editor for AltWalker and GraphWalker
- AltWalker Model Visualizer for VSCode
- Actions and Guards (from AltWalker's documentation)
- AltWalker examples (Python and C#/.NET)
- AltWalker CLI
- Port of PetClinic MBT example to AltWalker and Python (code for this tutorial)
- GraphWalker models for testing the PetClinic site (source-code)

- GraphWalker
- GraphWalker documentation pages