End-to-End Testing of COOLAPS Applications

Camilo Correa Restrepo, Jacques Robin[*]

December 17, 2021

Contents

1 Introduction

Experience has shown that, as with most engineered artifacts, systematically testing software is incredibly important in ensuring both its quality at a given instant and its ability to evolve without regressing as new features are added. To do this, several techniques have emerged in recent years to address these needs. Of particular importance to us is the notion of end-to-end testing. This type of testing, in essence, treats the software component, in its entirety, as a black box and ensures, instead, that its behavior, from an end-user perspective, conforms to the system's specification. To do this, instead of manually interacting with the system, particular user-flows are first designed and then translated into series of steps that can be performed and (crucially) checked by a test system in order to examine whether the

[*]camilo.correa-restrepo@univ-paris1.fr, jacques.robin@univ-paris1.fr

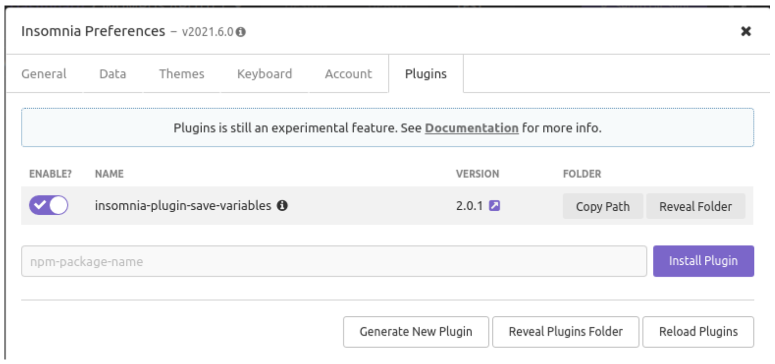
application performs as expected. This does not replace other testing done directly at the code-level of the application, instead, it is designed to enable developers to ensure that the application as a whole works as intended, whatever the internal behavior may be. While this type of testing is most commonly done with web and mobile applications with GUIs, nothing impedes one from utilizing test tooling to perform these tests on, in our case, web services exposed through REST APIs. The one difference would be that instead of determining which buttons to click or what text to enter into fields, one prepares sets of web requests linked by the responses of the application.

2 Tooling and Preparation

Whilst there are no hard and fast rules on which tools to use for end-to-end testing, what's most important is to have mechanisms to be able to automate the sending, reception and interpretation of responses so that user-workflows can be automated. Before one can begin performing these tests, and since we have no user interface that can guide us as to what elements we need to interact with, we require some documentation as to how our API endpoint works. The best way to do this is by having an API specification in a machine-readable format that both documents the API and its use and that helps us then program the tests that will be subsequently done. This can be achieved by using the OpenAPI standard found at `https://swagger.io/specification/`. This allows one to compile a complete specification of one's API in YAML describing inputs, outputs, payloads and so on. This specification is ideally done with a tool that supports the OpenAPI syntax, but this is not necessary. In our case, since our primary interest lies in REST API testing, we will be using Insomnia `https://insomnia.rest/` to import our specification and then generate the tests based on this implementation. In addition, we will need an insomnia plugin to be able to create our complex interactions with the endpoints. Once insomnia has been installed, we must install the plugin by:

1. going to the **Application** > **Preferences** menu;
2. on the **Plugins** tab writing "insomnia-plugin-save-variables";
3. clicking on the **Install Plugin** button;
4. restarting insomnia.

We must make sure the plugin tab looks as follows before continuing:

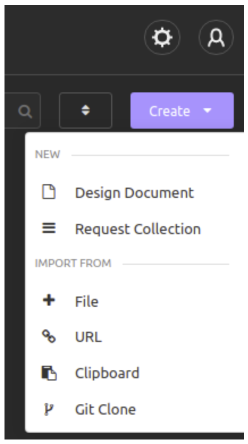

Insomnia Preferences – v2021.6.0

General | Data | Themes | Keyboard | Account | Plugins

Plugins is still an experimental feature. See Documentation for more info.

ENABLE?	NAME	VERSION	FOLDER
✓	insomnia-plugin-save-variables	2.0.1	Copy Path · Reveal Folder

The specification we will use will be for our Wumpus World Simulator available here: `https://github.com/kaiser185/wumpus_simul_ai4eu`. We will not go into details here as to how it functions: to make sense of all that is to follow it is **REQUIRED** to at the very least familiarize oneself with the README and the basic idea of the Wumpus world. It has been developed alongside the application and serves, as mentioned above as the principal documentation of the API. We will import it into insomnia through the **Create** > **Import from** > **File** Menu option as shown here:

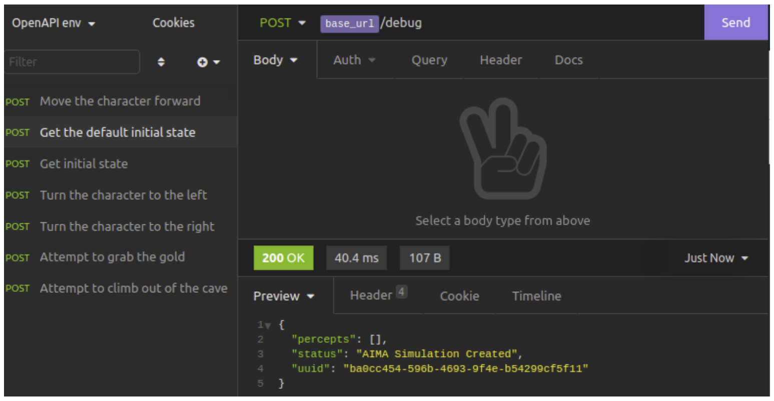


We will import it as a **Design Document** which will allow us to browse and even edit the specification on the fly, though that is beyond the scope of this tutorial. With the specification imported, we are now ready to begin creating our test suite.

3 Creating the requests

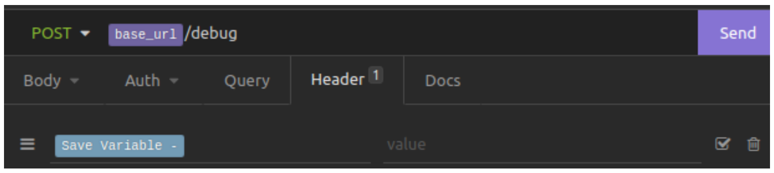
For the purposes of this tutorial, a special purpose *debug* endpoint has been created which will allow us to have far more control over our test environment. Since the simulation creates the environments at random given the parameters one gives it, and in order to have a set of tests that we can repeat at will and that we can control fully, we will use this debug endpoint to always give a world whose parameters we know already. This is important so we can fully imagine all interaction scenarios that a user might observe through normal use, without the uncertainty of randomly generated worlds requiring us to adapt our queries every time.

With the specification imported, we can move over to the **DEBUG** tab of our insomnia client to begin preparing our requests. Assuming the server is running, we can first test that our debug request successfully creates a default simulation session by clicking the **SEND** button as shown below:

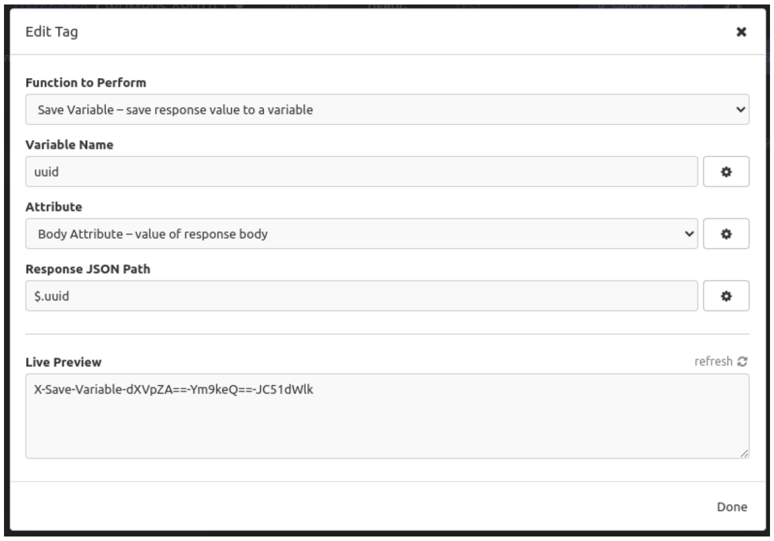

OpenAPI env
Cookies
Filter
POST Move the character forward
POST Get the default initial state
POST Get initial state
POST Turn the character to the left
POST Turn the character to the right
POST Attempt to grab the gold
POST Attempt to climb out of the cave
POST
base_url/debug
Send
Body
Auth
Query
Header
Docs
Select a body type from above
200 OK
40.4 ms
107 B
Just Now
Preview
Header 4
Cookie
Timeline

```
{
  "percepts": [],
  "status": "AIMA Simulation Created",
  "uuid": "ba0cc454-596b-4693-9f4e-b54299cf5f11"
}
```

Now, given that our server gives out unique ids for each simulation, if we want to use it to explore the same simulation and create a suite of tests that follow one another (to test the whole user's interaction and not just that the endpoint works, which we can do within this **DEBUG** tab), we need to have the subsequent requests reuse the same UUID as when the simulation is created. To achieve this, we must first create a **virtual header**, i.e. one that will not be sent with the request but will instead trigger the plugin (c.f. the plugin documentation at `https://insomnia.rest/plugins/insomnia-plugin-save-variables` for further explanation as to how this works). In this case, we must first add the **Save Variable** tag to the header. This is done by using the **CTRL+SPACE** key combination and selecting it from the options that appear; once done it should look like this:



Next, click on **Save variable** to open the configuration menu. Set the parameters as follows:



Edit Tag

Function to Perform
Save Variable – save response value to a variable

Variable Name
uuid

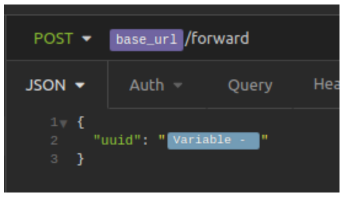
Attribute
Body Attribute – value of response body

Response JSON Path
$.uuid

Live Preview
X-Save-Variable-dXVpZA==-Ym9keQ==-JC51dWlk

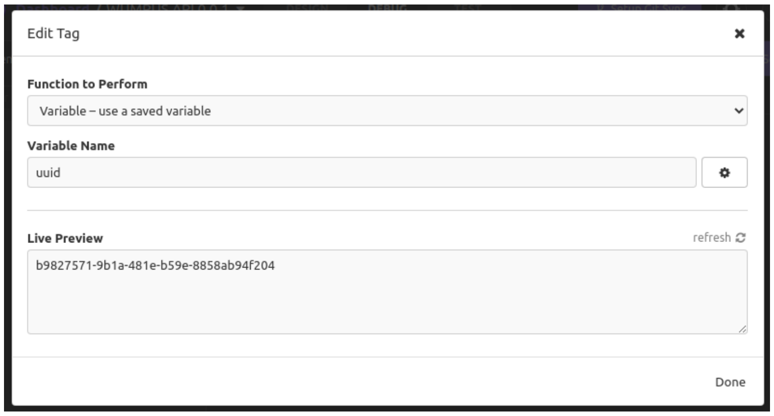
This will ensure that every time the `debug` endpoint is queried, a **uuid** variable will be set internally in insomnia, which we can then reuse for all our other requests.

Now, if we want to move forward one space in the simulation, we must first move over to our **"Move the character forward"** request. Since we imported the specification, and there we specified that the **uuid** parameter is a string, we must now modify it by first setting the string to be empty and then, using the **CTRL+SPACE** key combination, we select one of insomnia's special functions (specifically that of Variable, similarly to what we did above) to grab the UUID value from the `debug` request which should yield something to this effect:



```
{
  "uuid": "Variable"
}
```

As before, we must click the highlighted text to configure this as follows:


Edit Tag

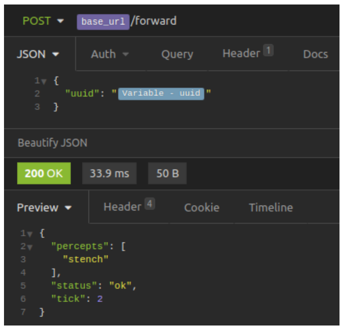
Function to Perform: Variable – use a saved variable

Variable Name: uuid

Live Preview

```
b9827571-9b1a-481e-b59e-8858ab94f204
```

If we test this new request we now see that the simulation has now advanced to the next tick and we observe the correct response, i.e., we see that the simulation tells us that we **smell** the Wumpus up ahead.


POST base_url/forward

```
{
  "uuid": "Variable - uuid"
}
```

200 OK 33.9 ms 50 B

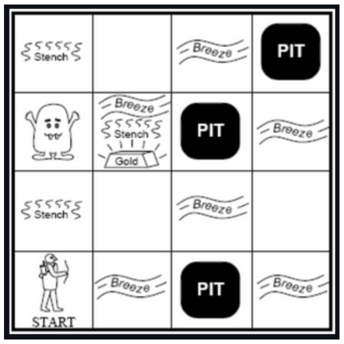
```
{
  "percepts": [
    "stench"
  ],
  "status": "ok",
  "tick": 2
}
```

We can now copy the JSON body of this request to all the other requests, since they all share the same body information, i.e. the UUID.

4 Building the automated test suite

Now that we have our requests configured, we can begin to create our automated tests suites. Within insomnia, and more generally within end-to-end testing, test suites for end to end testing are generally done by creating chains of unit tests. This allows us to isolate each "part" of the suite as a whole while still threading together a complex interaction. This is one of the fundamental differences between traditional unit testing and end-to-end testing.

We will begin by building one of the simplest possible interactions with simulator: we will start the simulator and immediately walk into the Wumpus and die. For reference, this is what our default environment looks like (note that the agent is facing the Wumpus by default):



To achieve such an interaction, we must walk forward twice. This means that we must do three requests to our simulator:

1. Start the simulation
2. Walk forward once
3. Walk forward a second time

To do this, we will program a set of tests using the test suite builder under Insomnia's **TEST** tab. Here we will use the *Chai* assertion library to write our tests (specifically in the *BDD* style) as described in the following links:

- `https://docs.insomnia.rest/insomnia/unit-testing`
- `https://www.chaijs.com/api/bdd/`

We will begin by creating a new test suite that we will appropriately call **Die** by clicking on the **New Test Suite** button. We will then use the **New Test** button to create our three tests in their appropriate order, selecting for each the suitable request: "Get the default initial state" for the first, "Move the character forward" for the other two. Once this is done, we can now code the actual assertions that we will test for within each unit test.

The first test will test for what we expect to see when the simulation starts, with the following code:

```
const response1 = await insomnia.send();
const body = JSON.parse(response1.data);
expect(response1.status).to.equal(200);
expect(body.status).to.equal("AIMA Simulation Created");
expect(body.percepts).to.be.empty;
```

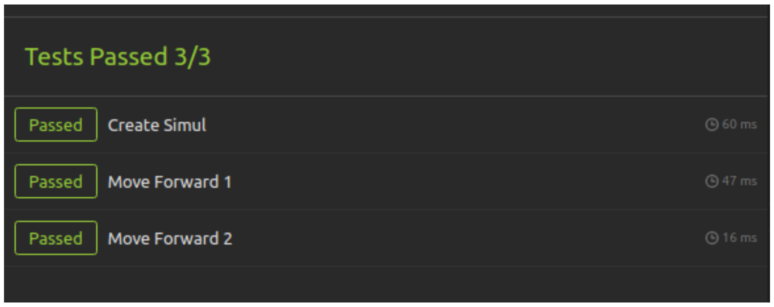
The second test will check that the simulation correctly interprets the action and returns the right percepts:

```
const response1 = await insomnia.send();
const body = JSON.parse(response1.data);
expect(response1.status).to.equal(200);
expect(body.status).to.equal("ok");
expect(body.percepts).to.include("stench");
expect(body.tick).to.equal(2);
```

The final test will check that the simulation correctly stops and detects that the agent has died when executing the action:

```
const response1 = await insomnia.send();
const body = JSON.parse(response1.data);
expect(response1.status).to.equal(200);
expect(body.status).to.equal("Simulation Halted - Agent Dead");
expect(body).to.not.have.property("tick");
expect(body).to.not.have.property("percepts");
```

Now, if we run the test suite, by clicking on **Run Tests**, we should see the following:



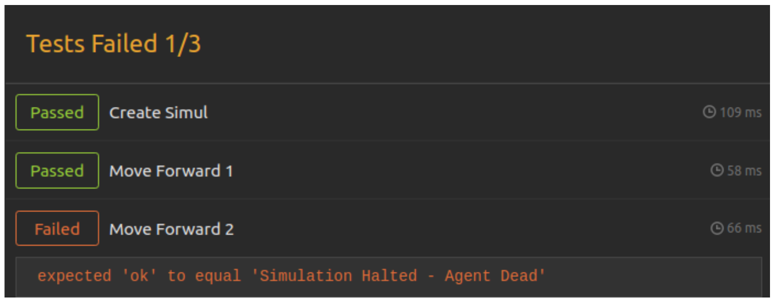
```
Tests Passed 3/3
Passed  Create Simul      60 ms
Passed  Move Forward 1    47 ms
Passed  Move Forward 2    16 ms
```

Indicating that we successfully managed to string together a complex end-to-end testing of our application.

5 Finding Bugs

The end-to-end testing approach is most important, of course, to ensure that a mostly feature-complete application can be used as expected. If one follows the test driven development approach, one should construct these test suites as one constructs the application, to, in a sense, guide application development. That said, it has some drawbacks, in particular related to the black-box view of the underlying components: one has relatively little information as to where faults come from, beyond the fact that the integration of all internal components is not correct in some way.

To illustrate this, and for the sake of example, let's imagine that a mistake exists in the code that performs the checks to stop the simulation when the agent dies. This would mean that when the agent takes his final, and very perilous, step into the Wumpus' square, the simulation would not register that the agent has died, and would allow the user to continue sending instructions when, in fact, the game would be over. If we were to run our test suite under such conditions we would now observe this:



```
Tests Failed 1/3

Passed   Create Simul        109 ms
Passed   Move Forward 1      58 ms
Failed   Move Forward 2      66 ms
  expected 'ok' to equal 'Simulation Halted - Agent Dead'
```

The system has not behaved as expected and something is clearly wrong with the internals of the application. Furthermore, we have a precise indication of **where** and at what **step** of a complex interaction with the system the fault is located. We can infer that the fault is probably related to either the checks for stopping the simulation, or the code that stops it itself. Unfortunately this is about as much information as can be gleaned from this testing approach in order to diagnose bugs: we can only get a general idea of where things are going wrong but not much more.

6 Conclusion

We have seen that end-to-end testing is a very powerful approach for evaluating software systems once they've been integrated, and, moreover, we observe that it enables one to truly test the system through the eyes of the end-user. While this approach certainly has its limitations, it remains an effective tool to ensure that software systems are stable as they evolve, whilst simultaneously avoiding the need to delve too deeply into the details of the application's internals. For applications that will be deployed as services, it should be considered a mandatory step in their development process. That being said, we must then complement this approach with other systematic testing methods, like unit testing, to be able to effectively fish out any bugs that may work themselves into our applications. In addition, though this goes beyond the scope of this tutorial, it is possible to fully automate this type of tests and integrate them into a Continuous Integration and Continuous Deployment (CI/CD) workflow.