Integration Testing Step Functions: Using sls-test-tools

Joel Hamilton · Follow
Published in Serverless Transformation · 5 min read · Mar 4, 2022

When building serverless applications, many people choose to use Step Functions to orchestrate work flows within their system. They are popular for their ability to configure direct service integrations, as well as manage failures, retries, parallelisation, and other features.

The states which make up a Step Function

A Step Function is a state machine, where states have types, namely:

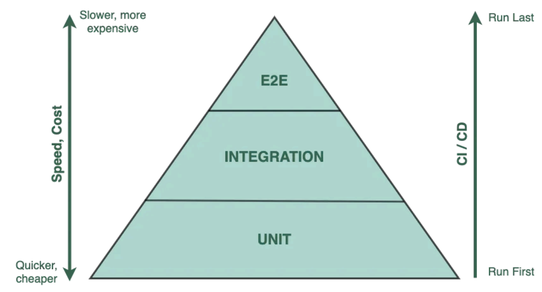
- **Task state:** for performing an operation
- **Choice state:** for choosing which branch of the state machine to execute next
- **Fail/Succeed state:** for ending the execution of a state machine
- **Pass state:** for passing its input to output, or injecting fixed data
- **Wait state:** for imposing a delay into the execution of the state machine
- **Parallel state:** for creating parallel branches of execution in the state machine
- **Map state:** for performing the same state action for each item in a list

Should we test Step Functions?

As with any AWS infrastructure, we need to be able to test it, so that we know we've configured it correctly and so that it can't be broken in the future. As discussed by Yan Cui, it's important that we test the logic in the individual states, as well as the overall state machine.

To test Step Functions, many people only test the individual states, and some choose not to test them at all. This is because these interactions can often be across logical areas of a system, and because the triggers and states are difficult to assert on without direct calls to the SDK — or extensive mocking. In any case, my advice is always to test, and in particular, to always test on the real infrastructure in order to best simulate the real production environment. I recommend spinning up infrastructure for testing purposes and tearing it down post-testing, so that costs are kept low. For more on this, see Ben Ellerby's article on Serverless Flow, a CI/CD branching workflow which does exactly that.

Unit vs Integration Testing



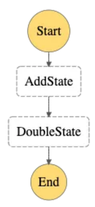
Unit tests focus on testing individual pieces of code, whereas integration tests check that various pieces of our system are interacting correctly with each other. It is important to have both!
Testing individual states is relatively simple, as one can just write checks on the individual pieces of logic in each state (e.g. test the output of a Lambda function is correct, or that a DynamoDB query returns the correct data, etc), with no mocking needed. These unit tests are often created through Test-Driven Development (where, incrementally, tests are written as part of the development cycle).

Integration tests are slightly more complex. You can use mocking libraries to simulate the behaviour of various AWS services, however these can be misleading, and require an overhead to keep updated. By using the real infrastructure, we eliminate the possibility of there being a case that mocks are unable to simulate, and avoid the need to constantly keep mocks updated.

Spinning up real infrastructure to test on is cheap as serverless services are billed on a pay-per-use basis. Given the preference for testing on real infrastructure, I've worked with my team to add Step Function assertions to sls-test-tools, the open source integration testing tool for AWS infrastructure maintained by aleios.

Introducing Step Functions assertions with sls-test-tools

The below code samples will demonstrate how to test a step function called 'TestAddAndDouble' which contains two task states, one which sums two numbers, and another which doubles the number.



Assert that the state machine has the correct execution status

First let's test that we get to a Succeeded status when we pass a valid input. To do this we can use the new **toHaveCompletedExecutionWithStatus()** assertion. This allows us to type in the name of the state machine we've executed (manually, or by triggering it with another event, e.g. hitting an API gateway endpoint, or triggering a Lambda function), as well as the expected status for the execution, (e.g. 'SUCCEEDED', 'FAILED', 'TIMED_OUT', etc). It then checks that the most recent execution of the state machine has the expected status (note that these tests should not be run in parallel). Here is an example of the usage of this assertion:

Assert that the state machine produces the correct output

Now let's check we get the right output given a valid input, for example if we enter {3, 6} we expect to get 18 as the output.
We can once again use a new assertion, the toMatchStateMachineOutput assertion. This allows us to type in the name of the state machine we've executed, as well as the expected output for this execution, and checks that the most recent execution of the state machine produced the expected output. Here is an example of the usage of this assertion:

Helper: Execute the Step Function until its completed

You'll have noticed above that we had to rely on the step function having been executed. Rather than require a manual step or complex call to the API we've also bundled in a helper function to simplify execution and wait for execution to complete. Here is an example of the call to the helper:

Putting it all together

Now if we combine the use of this helper, and our two assertions we can see a mature test suite for our TestAddAndDouble Step Function.

Conclusion

With Serverless, integration testing against the real services is more important than ever, and Step Functions are far too often neglected!
sls-test-tools now provides a helper which allows us to execute a Step Function until completion, and two new assertions to check that the Step Function is configured correctly (i.e. it provides the right output for a variety of inputs, and consistently completes with the correct execution status).
Check out sls-test-tools here!

> *P.S. sls-test-tools is now typed, so it can be used in your next Typescript project!*

Serverless-Transformation is an **aleios** initiative. **aleios** helps startups disrupt and large organisations to remain competitive using the best of Cloud-Native, **Serverless**.