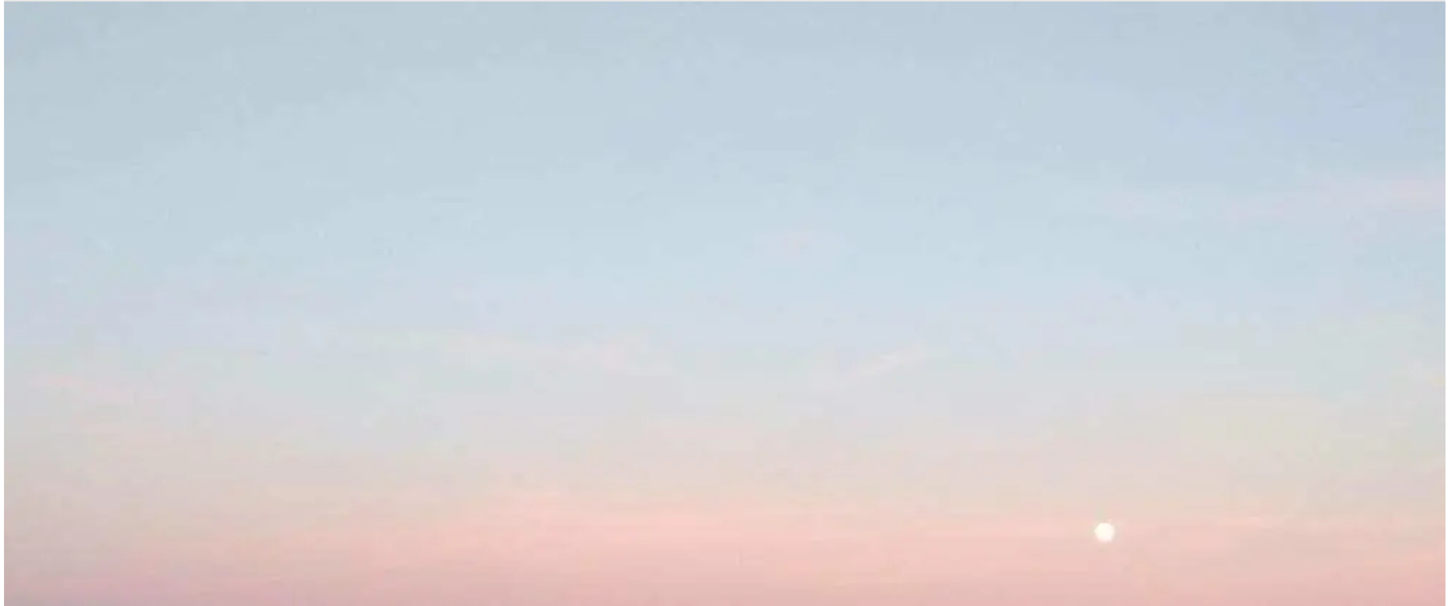


Wojciech Matuszewski for AWS Community Builders
Posted on 14 Feb

Testing AWS Step Functions flows

#aws #serverless #typescript #cdk

AWS Step Functions is a serverless orchestrator service. Since its inception, it has become a go-to for all my low-code and orchestration needs on AWS.

Recently, AWS announced that the Step Functions Local can now mock service integrations. I found the announcement a great opportunity and excuse to revisit the topic of testing in the context of AWS Step Functions, which historically was a bit hard.

This article will cover the techniques I use for testing flows that utilize the AWS Step Functions service as the logic orchestrator. Let us begin.

> The code in this blog post is written in TypeScript and uses AWS CDK for infrastructure piece. This GitHub repository contains all the code used in this article.

WojciechMatuszewski / **testing-step-functions**

Testing AWS Step Functions flows

This repository contains examples of how one might test various AWS Step Functions flows.

The test files are located in the `lib/__tests__` directory.

Deployment

1. `npm run boostrap`
2. `npm run deploy`

Running the tests

1. Ensure that Docker is running.
2. Pull the *aws-stepfunctions-local* image.
3. `npm run test`

View on GitHub

Layers of testing and confidence

Before we dive into specific techniques, we shall take a swift detour and talk about testing in general.

There are many heuristics when it comes to testing. There is the classical testing pyramid or the testing honeycomb to name a few. In my personal opinion, **in the context of serverless applications**, tests written according to the testing honeycomb will give you much more confidence and ROI per test written than the "traditional" testing pyramid approach.

Before you base most of your tests on the local implementation of an AWS service, I urge you to think about the confidence the test gives you and less about how easy it is to write. Give the testing honeycomb a try. I'm positive you will not regret it!

By writing integration / end-to-end tests

By utilizing e2e / integration tests, we gain the most confidence from our tests. That said, the tests are usually slow and can be hard to maintain to some degree.

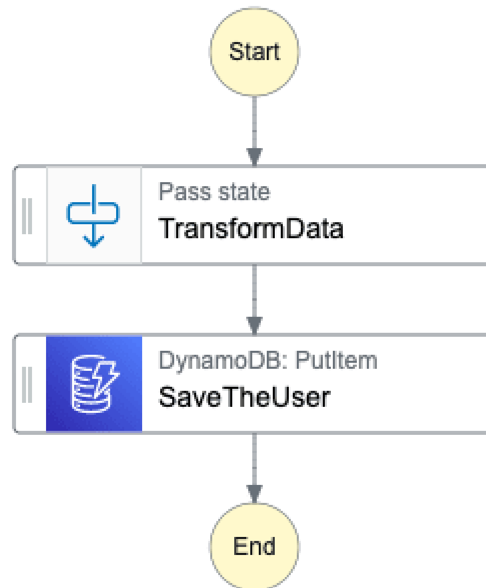
Here, you will be interacting with AWS services directly, executing the AWS Step Function and asserting the side-effects of the flow. The side-effect can be, for example, a new item in Amazon DynamoDB or a message pushed to Amazon SQS.

This style of testing is not free of challenging problems to tackle. Testing AWS Step Functions flow end-to-end becomes tricky when the Step Function definition is complex and contains multiple branches and error fallbacks.

Let us start with a basic example of saving a user into the Amazon DynamoDB table, then work our way up with Step Function definition complexity.

Simplistic Step Functions workflows

The following image represents the Step Function definition we would like to test.



Lacking branching logic and error handling, all we have to do is test a single execution path.

> You should most likely always have error handling in place in your Step Function definition. This particular Step Function is only here for demonstration purposes. We will tackle testing error states later in the article.

Here is how I would write the test I'm referring to.

```
// simplistic-e2e.test.ts

import { SFNClient, StartExecutionCommand } from "@aws-sdk/client-sfn";

const sfnClient = new SFNClient({});

test("Saves the user in the DynamoDB table", async () => {
  const startExecutionResult = await sfnClient.send(
    new StartExecutionCommand({
      stateMachineArn: process.env.SIMPLISTIC_E2E_STEP_FUNCTION_ARN,
      input: JSON.stringify({
        firstName: "John",
        lastName: "Doe"
      })
    })
```

```
    );

    await expect({
      region: process.env.AWS_REGION,
      table: process.env.SIMPLISTIC_E2E_DATA_TABLE_NAME
    }).toHaveItem(
      {
        PK: `USER#${startExecutionResult.executionArn}`
      },
      {
        PK: `USER#${startExecutionResult.executionArn}`,
        firstName: "John",
        lastName: "Doe"
      }
    );
  }, 15_000);
```

First, I start the Step Function, and then I assert, using the [aws-testing-library](), whether the item was correctly saved into the Amazon DynamoDB.

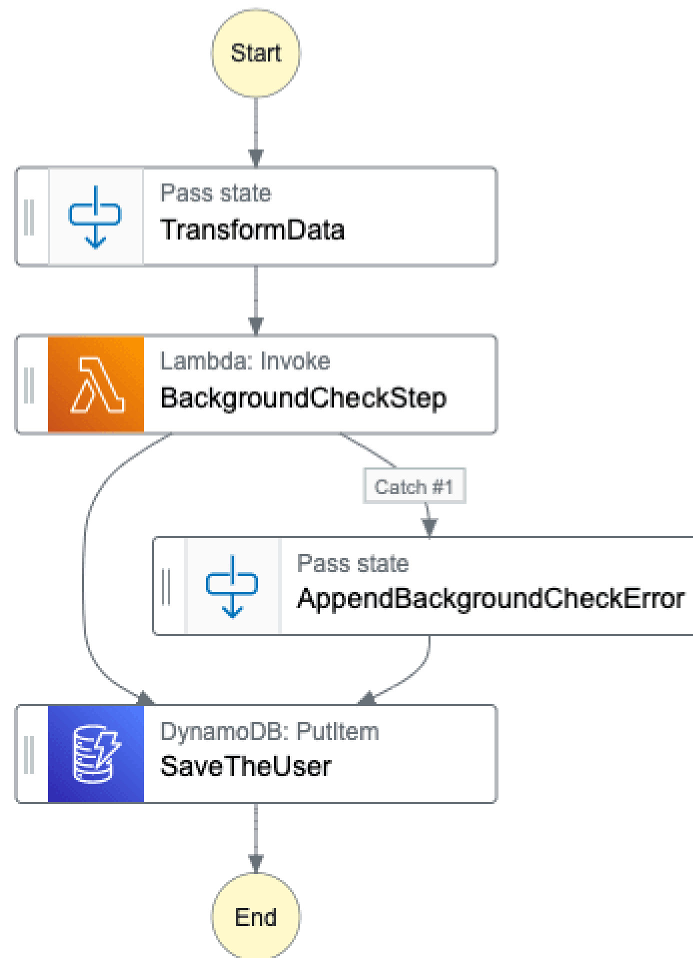
> Check out the [sls-test-tools]() as well. It is a great library. I'm using *aws-testing-library* because I'm used to it.

And that is it! Since this Step Function did not contain multiple branches and error handling, writing an end-to-end test takes little to no effort. With the most basic example behind us, let us tackle error states and multiple branches next.

Step Functions with branching logic

Step Functions can be complex, especially when taking error handling and `Choice` steps into account. With the increased complexity comes increased difficulty in writing end-to-end tests.

Let us evolve our Step Function to include "background-check" [AWS Lambda]() function. Depending on the result, the user in the DynamoDB table will have its `backgroundCheck` attribute populated (either "PASS", "FAIL" or "ERROR").



> You can find the AWS Step Function AWS CDK definition here.

To test all possible execution paths of the Step Function, **we have to have a way to force an error or particular response** for the `BackgroundCheckStep` AWS Lambda function – not an easy task!

So what can we do in this situation? Enter **aws-stepfunctions-local**.

The *aws-stepfunctions-local* is a local implementation of the AWS Step Functions service provided to us by great folks at AWS. With this tool, we **will force the lambda to error without mocking other steps**.

> You could write a test that executes asserts on the status returned by the "background-check" AWS Lambda function. For the interest of time, I've chosen only to write a test for the case where the "background-check" fails.

Since I will be using the Docker version of the *aws-stepfunctions-local*, the **first step** is to integrate the act of spinning up and spinning down the container into the testing flow. My personal go-to in such situations is the testcontainers package.

```
// branching-logic-e2e.test.ts

let container: StartedTestContainer | undefined;

beforeAll(async () => {
  const mockConfigPath = join(__dirname, "./branching-logic-e2e.mocks.json");
  container = await new GenericContainer("amazon/aws-stepfunctions-local")
    .withExposedPorts(8083)
    .withBindMount(mockConfigPath, "/home/branching-logic-e2e.mocks.json", "ro'
    .withEnv("SFN_MOCK_CONFIG", "/home/branching-logic-e2e.mocks.json")
    .withEnv("AWS_ACCESS_KEY_ID", process.env.AWS_ACCESS_KEY_ID as string)
    .withEnv(
      "AWS_SECRET_ACCESS_KEY",
      process.env.AWS_SECRET_ACCESS_KEY as string
    )
    // For federated credentials (for example, SSO), this environment variable
    .withEnv("AWS_SESSION_TOKEN", process.env.AWS_SESSION_TOKEN as string)
    .withEnv("AWS_DEFAULT_REGION", process.env.AWS_REGION)
    .start();
}, 15_000);

afterAll(async () => {
  await container?.stop();
}, 15_000);
```

Let us unpack what is going on here.

First, the mysterious `mockConfigPath` and subsequent usages of this variable. The path points to a mock configuration file described on [aws-stepfunctions-local documentation page](#) and contains a mock definition for the `BackgroundCheckStep`.

```
{
  "StateMachines": {
    "BranchingLogic": {
      "TestCases": {
        "ErrorPath": {
          "BackgroundCheckStep": "BackgroundCheckError"
        }
      }
    }
  },
  "MockedResponses": {
    "BackgroundCheckError": {
      "0": {
        "Throw": {
```

```
          "Error": "Lambda.TimeoutException",
          "Cause": "Lambda timed out."
        }
      }
    }
  }
}
```

Notice that I'm only concerned with the `BackgroundCheckStep` here. **By only mocking `BackgroundCheckStep`, I can force this particular step to fail**. Other steps are not mocked. Thus **the *aws-stepfunctions-local* will reach out to native AWS services** – in our case Amazon DynamoDB.

The Docker image needs to have AWS-related environment variables populated to allow *aws-stepfunctions-local* to talk to other AWS services. Refer to [this documentation page] for more information.

```
.withEnv("AWS_ACCESS_KEY_ID", process.env.AWS_ACCESS_KEY_ID as string)
.withEnv("AWS_SECRET_ACCESS_KEY",process.env.AWS_SECRET_ACCESS_KEY as string)
.withEnv("AWS_SESSION_TOKEN", process.env.AWS_SESSION_TOKEN as string)
.withEnv("AWS_DEFAULT_REGION", process.env.AWS_REGION)
```

As for the test itself, the first thing to do is gather necessary information about the Step Function under test.

```
// branching-logic-e2e.test.ts

test("Handles the failure of the BackgroundCheck step", async () => {
  const sfnClient = new SFNClient({});

  const describeStepFunctionResult = await sfnClient.send(
    new DescribeStateMachineCommand({
      stateMachineArn: process.env.BRANCHING_LOGIC_E2E_STEP_FUNCTION_ARN
    })
  );
  const stepFunctionDefinition =
    describeStepFunctionResult.definition as string;
  const stepFunctionRoleARN = describeStepFunctionResult.roleArn as string;

  // Rest of the test...
}, 50_000);
```

We will need all of this information since we will be re-creating this Step Function locally. Remember, the *aws-stepfunctions-local* is the engine that runs our Step

Function.

Next, let us re-create and run the Step Function that lives in the cloud using the *aws-stepfunctions-local*.

```
// branching-logic-e2e.test.ts

// Test setup ...

test("Handles the failure of the BackgroundCheck step", async () => {
  // Previous code snippet ...

  const sfnLocalClient = new SFNClient({
    endpoint: `http://localhost:${container?.getMappedPort(8083)}`
  });

  const createLocalSFNResult = await sfnLocalClient.send(
    new CreateStateMachineCommand({
      definition: stepFunctionDefinition,
      name: "BranchingLogic",
      roleArn: stepFunctionRoleARN
    })
  );

  const startLocalSFNExecutionResult = await sfnLocalClient.send(
    new StartExecutionCommand({
      stateMachineArn: `${
        createLocalSFNResult.stateMachineArn as string
      }#ErrorPath`,
      input: JSON.stringify({
        firstName: "John",
        lastName: "Doe"
      })
    })
  );

  // Rest of the test...
}, 50_000);
```

There are two **essential** pieces of detail I would like to bring your attention to.

1. The `name` of the Step Function I'm creating. The **`name` parameter must be the same as declared in the mock configuration file**.

2. The `stateMachineArn` format in the `StartExecutionCommand` call. The convention of `ARN#TEST_CASE` is required by *aws-stepfunctions-local*. Refer to this documentation

page to learn more.

All that is left is to assert on the data of a given Amazon DynamoDB item. The assertion is almost identical as in the *Simplistic Step Functions* section.

```
// branching-logic-e2e.test.ts

// Test setup ...

test("Handles the failure of the BackgroundCheck step", async () => {
  // Previous code snippet ...

  await expect({
    region: process.env.AWS_REGION,
    table: process.env.BRANCHING_LOGIC_E2E_DATA_TABLE_NAME
  }).toHaveItem(
    {
      PK: `USER#${startLocalSFNExecutionResult.executionArn}`
    },
    {
      PK: `USER#${startLocalSFNExecutionResult.executionArn}`,
      firstName: "John",
      lastName: "Doe",
      // The `BackgroundCheckStep` must have failed for this attribute to have
      backgroundCheck: "ERROR"
    }
  );
}, 50_000);
```

We assert on an Amazon DynamoDB table that lives in the AWS. Since we **did not** provide any mocks for the step that saves the user data to Amazon DynamoDB, the *aws-stepfunctions-local* made the request to the actual service, utilizing the credentials from Docker environment variables.

▶ Click to expand (the whole test definition)

By decomposing AWS Step Function Tasks

Switching gears from a somewhat complex end-to-end tests world, there exists another technique I would like to highlight.

I first saw this method of testing various AWS Step Function tasks while browsing code written by my colleagues at Stedi.

My friend Graham Allan wrote about it on his blog here. Check it out!

Testing Pass state transformations

Have you ever spent way too much time trying to transform data via the `Pass` state to the shape you need it to be? I know I have.

As great as the AWS console and the AWS Step Functions data flow simulator is, I find the feedback loop of a test case unbeatable.

So, how can we reliably extract those `Pass` states from our AWS CDK code and test them? Here is my solution.

Here is our sample *Construct* definition. It contains the `transformDataStep` that we would like to test.

```
// pass-states-integration.ts

export class PassStatesIntegration extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);

    const transformDataStep = new aws_stepfunctions.Pass(
      this,
      "TransformDataStep",
      {
        parameters: {
          payload: aws_stepfunctions.JsonPath.stringAt(
            "States.Format('{} {}', $.firstName, $.lastName)"
          )
        }
      }
    );

    // You can imagine the definition being a bit more complex.
    const stepFunctionDefinition = transformDataStep;

    const stepFunction = new aws_stepfunctions.StateMachine(
      this,
      "StepFunction",
      {
        definition: stepFunctionDefinition
      }
    );
  }
}
```

We will be using the *aws-stepfunctions-local*, so the test setup looks very similar to the previous code snippets in this article.

```
// pass-states-integration.test.ts

let container: StartedTestContainer | undefined;

beforeAll(async () => {
  container = await new GenericContainer("amazon/aws-stepfunctions-local")
    .withExposedPorts(8083)
    .withEnv("AWS_DEFAULT_REGION", process.env.AWS_REGION)
    .start();
}, 15_000);

afterAll(async () => {
  await container?.stop();
}, 15_000);
```

To test the `transformDataStep`, we must retrieve the `transformDataStep` ASL definition. We can do this in two ways.

1. Use the `DescribeStateMachine` API call, like we did in the *Step Functions with branching logic* section.
2. Make the `transformDataStep` a publicly accessible property on the `PassStatesIntegration` construct.

I'm going to go with option number two since we have already seen option number one in action.

```
// pass-states-integration.ts

export class PassStatesIntegration extends Construct {
+ public transformDataStep: aws_stepfunctions.Pass;

  constructor(scope: Construct, id: string) {
    super(scope, id);

-   const transformDataStep = new aws_stepfunctions.Pass(
+   this.transformDataStep = new aws_stepfunctions.Pass(
      this,
      "TransformIncomingDataStep",
      {
        parameters: {
          payload: aws_stepfunctions.JsonPath.stringAt(
```

```
            "States.Format('{} {}', $.firstName, $.lastName)"
          )
        }
      }
    );

    // You can imagine the definition being a bit more complex.
-   const stepFunctionDefinition = transformDataStep;
+   const stepFunctionDefinition = this.transformDataStep;

    const stepFunction = new aws_stepfunctions.StateMachine(
      this,
      "StepFunction",
      {
        definition: stepFunctionDefinition
      }
    );
  }
}
```

With the `transformDataStep` made public, the test body would look as follows.

```
// pass-states-integration.test.ts

// Test setup ...

test("Handles the failure of the BackgroundCheck step", async () => {
  const stack = new cdk.Stack();
  const construct = new PassStatesIntegration(stack, "PassStatesIntegration");

  const transformDataStepDefinition = construct.transformDataStep.toStateJson()
  const stepFunctionDefinition = JSON.stringify({
    StartAt: "TransformDataStep",
    States: {
      TransformDataStep: {
        ...transformDataStepDefinition,
        End: true
      }
    }
  });

  const sfnLocalClient = new SFNClient({
    endpoint: `http://localhost:${container?.getMappedPort(8083)}`
  });

  const createLocalSFNResult = await sfnLocalClient.send(
    new CreateStateMachineCommand({
```

```
      definition: stepFunctionDefinition,
      name: "PassStates",
      roleArn: "arn:aws:iam::012345678901:role/DummyRole"
    })
  );

  const startLocalSFNExecutionResult = await sfnLocalClient.send(
    new StartExecutionCommand({
      stateMachineArn: createLocalSFNResult.stateMachineArn,
      input: JSON.stringify({
        firstName: "John",
        lastName: "Doe"
      })
    })
  );

  await waitFor(async () => {
    const getExecutionHistoryResult = await sfnLocalClient.send(
      new GetExecutionHistoryCommand({
        executionArn: startLocalSFNExecutionResult.executionArn
      })
    );

    const successState = getExecutionHistoryResult.events?.find(
      event => event.type == "ExecutionSucceeded"
    );

    expect(successState?.executionSucceededEventDetails?.output).toEqual(
      JSON.stringify({ payload: "John Doe" })
    );
  });
}, 20_000);
```

The ASL for the `TransformDataStep` is extracted via the `toStateJson` method. The rest of the test is similar to how we did it previously. The only difference is how we make the assertion.

This testing method is analogous to the one described by the *By decomposing AWS Step Function Tasks* section.

Closing words

I hope you find this blog post helpful regarding AWS Step Functions testing.

Consider following me on Twitter for more serverless content - [@wm_matuszewski](https://twitter.com/wm_matuszewski).

Thank you for your valuable time.

Discussion (0)

Code of Conduct • Report abuse

AWS Community Builders

Build On!

Would you like to become an AWS Community Builder? Learn more about the program and apply to join when applications are open next.

Learn more

More from AWS Community Builders

AWS Cloud WAN: The General Availability and Product Features
#aws #cloud #cloudwan #networking

There will be 175 Zettabytes of data in the world by 2025. Where will we store it?
#awsdatabases #terraform #bigdata #aws

AWS Cost Explorer - Cost Anomaly Detection Report identified an unauthorized Amazon Sagemaker Canvas user
#aws #beginners #security #cost