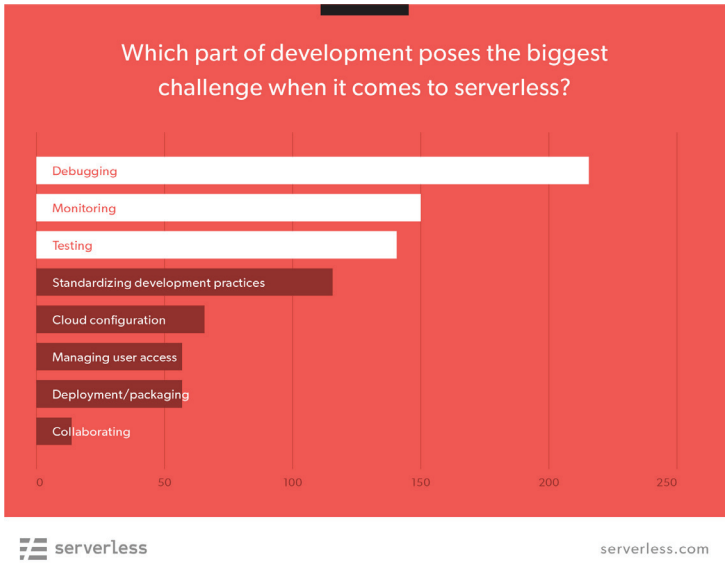
ROOKOUT

The Comprehensive Guide to Serverless Debugging

Contents

Introduction:

Software debugging is an essential capability for developers, but debugging serverless architectures requires a dedicated approach. According to a survey conducted by Serverless Inc., debugging, monitoring and testing are the top challenges when it comes to serverless, with debugging being the most common challenge.



By the end of this guide, I hope to have reduced some of the pain of debugging serverless functions.

Serverless functions are, by design, very hard to attach anything to. As a result, we're witnessing new solutions for debugging serverless, both locally and remotely. In this guide, I present several solutions for debugging your functions, along with basic usage information. For each solution, I also provide references for additional options and specifications.

Before getting into the technical solutions, it's worth taking a look at the first basic decision: whether to debug your serverless functions locally or in production.

Local debugging

Debugging serverless functions locally is different than debugging traditional applications that can be run locally. Local serverless debugging requires the cloud provider's entire environment to be emulated (AWS Lambda, IBM Openwhisk, Google Cloud Functions etc…) in order to accurately reproduce the desired application flow.

Several solutions provide emulation for serverless environments. Currently, local debugging solutions are applicable to specific languages, frameworks and cloud providers.

Pros:

- Easy to set up
- Doesn't interfere with production
- Debugging a serverless session using breakpoints grants full control of the application flow
- You can extract functions using custom events, on-demand

Cons:

- Doesn't represent the true production environment and resources, including elements missing from the simulation: events, network, databases
- Almost impossible to trace back production issues locally
- Limited support for languages, frameworks and cloud providers
- No standard for debugging locally
- Tool inflation: each solution provides a very narrow technological stack (specific language, specific framework and specific cloud provider)

Production debugging

Serverless environments are extremely “production-first”, and there’s no ideal way to actually execute them locally. As discussed above, there are emulators for running particular serverless environments locally, but they’re far from perfect.

Additionally, there are a few more unique challenges:

1. Serverless by nature is event-driven. This makes debugging harder as minor differences in the event content, format, or ordering can make big differences. Simulating those events is potentially a **big** difference that can make it harder to reproduce and test.
2. Serverless by nature is stateless. This makes serverless a heavy consumer of external datastores, which often provided by the cloud provider (sometimes using proprietary technology). Those datastores create another big difference between development and production.

When facing a bug, reproducing it reliably can be half the battle. In many data-driven environments, the main factor in actually reproducing the bug is getting the right input data. Migrating data from production to development can easily take hours (or even days) and may be hampered by security, compliance and other considerations. Production debugging tools allow developers to study the bug ‘in the wild’ in production without wasting time and resources on data migration, and without exposing sensitive data.

Pros

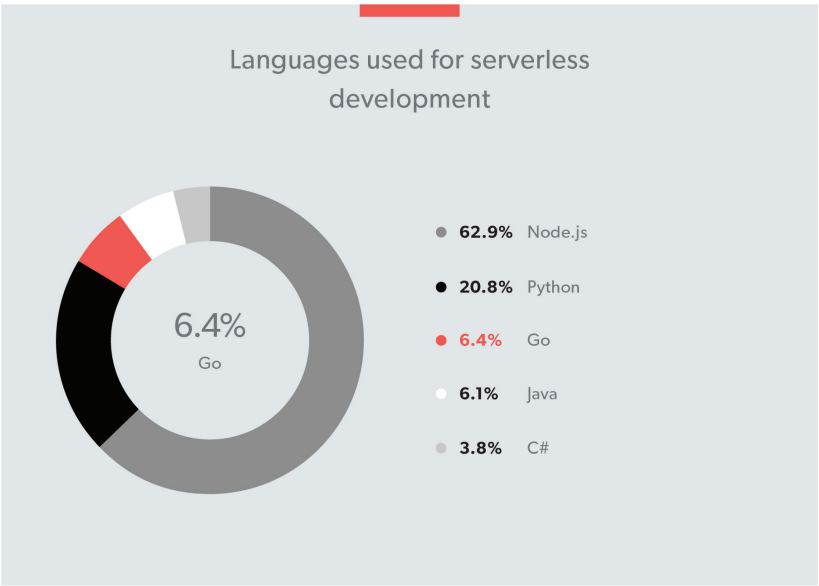
- Once a production debugging solution is integrated within your environment, you are ready to investigate issues rapidly
- Skip CI/CD for rapid debugging sessions
- Debugging production does not have overhead on your production
- Easy to trace requests among multiple microservices \ serverless functions
- Current solutions provide a holistic solution for most languages, frameworks and cloud providers
- New production debugging solutions allow you to perform collaborative debugging

Cons:

- Setup requires new code deployment
- Learning curve for using edge technologies for production debugging
- Unable to actually set breakpoint, although some solutions provide a fully descriptive dump frame just like a local debugger would do

A Note on Languages:

Node.js is king when it comes to serverless. Serverless Inc's survey showed that Node.js is used by 63% of all serverless developers, with Python coming next at 21%. Although this guide will cover a range of debugging options, it will go into more detail for Node.js (and occasionally Python) because those are the languages being used.



Serverless Framework

The Serverless Framework from Serverless Inc. allows you to run and debug serverless functions locally. It basically emulates the relevant environment and simulates the context with simple mock data. Serverless Framework works with various cloud providers and languages.

AMAZON WEB SERVICES (AWS LAMBDA):

The following command will trigger your desired function locally:
serverless invoke local --function functionName

These options allow you to control the entire function invocation:

- --function or -f The name of the function in your service that you want to invoke locally. Required.
- --path or -p The path to a json file holding input data to be passed to the invoked function as the event. This path is relative to the root directory of the service.
- --data or -d String data to be passed as an event to your function. Keep in mind that if you pass both --path and --data, the data included in the --path file will overwrite the data you passed with the --data flag.
- --raw Pass data as a raw string even if it is JSON. If not set, JSON data are parsed and passed as an object.
- --contextPath or -x, The path to a json file holding input context to be passed to the invoked function. This path is relative to the root directory of the service.
- --context or -c, String data to be passed as a context to your function. Same like with --data, context included in --contextPath will overwrite the context you passed with --context flag.

See this for invocation examples using the serverless framework on AWS:
https//:serverless.com/framework/docs/providers/aws/cli-reference/invoke-local/

At present, Serverless Framework only supports local invocation of Node.js, Python & Java functions for AWS Lambda.

GOOGLE CLOUD PLATFORM (GOOGLE CLOUD FUNCTIONS)

The following command triggers a local function on Google Cloud Platform:
serverless invoke local -f functionName

These options allow you to control the entire function invocation:

- --function or -f The name of the function in your service that you want to invoke. Required. _ --data or -d Data you want to pass into the function
- --path or -p Path to JSON or YAML file holding input data. This path is relative to the root directory of the service.
- --raw Pass data as a raw string even if it is JSON. If not set, JSON data are parsed and passed as an object.
- --contextPath or -x, The path to a json file holding input context to be passed to the invoked function. This path is relative to the root directory of the service.
- --context or -c, String data to be passed as a context to your function. Same like with --data, context included in --contextPath will overwrite the context you passed with --context flag.

See this for invocation examples using the serverless framework on Google cloud platform:
https//:serverless.com/framework/docs/providers/google/cli-reference/invoke-local/

APACHE OPENWHISK:

The following command triggers a local function on Apache Openwhisk:

serverless invoke local --function functionName

These options allow you to control the entire function invocation:

- --function or -f The name of the function in your service that you want to invoke locally. Required.
- --path or -p The path to a json file holding input data to be passed to the invoked function. This path is relative to the root directory of the service. The json file should have event and context properties to hold your mocked event and context data.
- --data or -d String data to be passed as an event to your function. Keep in mind that if you pass both --path and --data, the data included in the --path file will overwrite the data you passed with the --data flag.

See this for invocation examples using the serverless framework on Apache Openwhisk:
https//:serverless.com/framework/docs/providers/openwhisk/cli-reference/invoke-local/

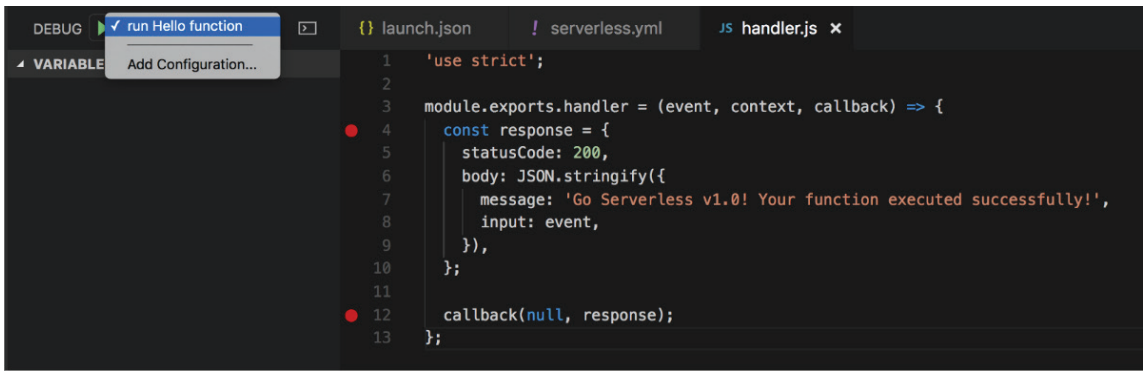
At present, Serverless Framework only supports local invocation of NodeJs and Python runtimes for Apache Openwhisk.

IDE setup for Serverless Framework

VISUAL STUDIO CODE

Using VS Code for debugging Node.js functions in Serverless Framework requires the following configuration under launch.json.

```
{
        "type": "node",
        "request": "launch",
        "name": "run myFunction",
        "program": "${workspaceFolder}/node_modules/ .bin/sls",
        "args":[
            "invoke",
            "local",
            "-f",
            "myFunction",
            "--data",
            "{}"
        ]
    }
```



After setting up the configuration, choose “run my function” from the drop down list and you are ready to debug!

WEBSTORM

Debugging Node.js functions using the Webstorm IDE is very simple. All you have to do is to add a new Node.js configuration with the name _<YOUR_FUNCTION’S_NAME>_

Under Node.js parameters put the path for the sls and your function’s name as follows:
<PATH_TO_SLS_MODULE> invoke local -f <YOUR_FUNCTION’S_NAME> -p

That's all!

SAM Local

Amazon Web Services's SAM Local is a local debugging option specifically created for AWS Lambda.
The following guide is taken from Amazon's documentation:*

The AWS Serverless Application Model (SAM) Command Line Interface (CLI), also referred to as SAM Local, lets you locally build, test, and debug serverless applications defined by AWS SAM templates. You can use the SAM init command to initialize a ready-to-deploy SAM application in your preferred runtime.

Installing SAM Local -
`$ pip install aws-sam-cli`

SAM CLI also offers the sam init command, which, when run, provides a fully-functional SAM application that you can use to further your understanding of the SAM application model or enhance your application to meet production needs. For more information, see Create a Simple App (sam init).

SAM CLI supports the following AWS runtimes:
1. node.js 4.3
2. node.js 6.10
3. node.js 8.10
4. python 2.7
5. python 3.6
6. java8
7. go 1.x
8. .NET 2.1

SAM CLI consists of the following CLI operations:

- **start-api:** Creates a local HTTP server hosting all of your Lambda functions. When accessed by using a browser or the CLI, this operation launches a Docker container locally to invoke your function. It reads the CodeUri property of the AWS::Serverless::Functionresource to find the path in your file system containing the Lambda function code. This path can be the project's root directory for interpreted languages like Node.js or Python, a build directory that stores your compiled artifacts, or for Java, a .jar file.
- If you use an interpreted language, local changes are made available within the same Docker container. This approach means you can reinvoke your Lambda function with no need for redeployment. For compiled languages or projects requiring complex packing support, we recommend that you run your own build solution and point AWS SAM to the directory that contains the build dependency files needed.
- **invoke:** Invokes a local Lambda function once and terminates after invocation completes.

```
# Invoking function with event file
$ sam local invoke "Ratings" -e event.json

# Invoking function with event via stdin
$ echo '{"message": "Hey, are you there?" }' | sam local invoke "Ratings"

# For more options
$ sam local invoke --help
```

To execute your Lambda function locally in debug mode, use the sam local invoke command with the -d option.

- **start-lambda:** Starts a local endpoint that emulates the AWS Lambda service's invoke endpoint, and you can invoke it from your automated tests. Because this endpoint emulates the Lambda service's invoke endpoint, you can write tests once and run them (without any modifications) against the local Lambda function or against a deployed Lambda function. You can also run the same tests against a deployed SAM stack in your CI/CD pipeline. For example;
  - **Start the local Lambda endpoint:** Start the local Lambda endpoint by running the following command in the directory that contains your AWS SAM template:
  - sam local start-lambda
  - This command starts a local endpoint at http://127.0.0.1:3001 that emulates the AWS Lambda service, and you can run your automated tests against this local Lambda endpoint. When you send an invoke to this endpoint using the AWS CLI or SDK, it will locally execute the Lambda function specified in the request and return a response. Invoke
  - **Run integration tests against the local Lambda endpoint:** In your integration testing, you can use the AWS SDK to invoke your Lambda function with test data, wait for a response and assert that the response is what you expect. To run the integration test locally, you should configure the AWS SDK to send the Invoke API call to the local Lambda endpoint started in the previous step. The following is a Python example (AWS SDKs for other languages have similar configurations):

```
- import boto3

  # Set "running_locally" flag if you are running the integration test locally
  if running_locally:

      # Create Lambda SDK client to connect to appropriate Lambda endpoint
      lambda_client = boto3.client('lambda',
```

```
                endpoint_url="http://127.0.0.1:3001",
                use_ssl=False,
                verify=False,
                config=Config(signature_version=UNSIGNED,
                        read_timeout=0,
                        retries={'max_attempts': 0}))
    else:
        lambda_client = boto3.client('lambda')


    # Invoke your Lambda function as you normally usually do. The function will run
    # locally if it is configured to do so
    response = lambda_client.invoke(FunctionName="HelloWorldFunction")

    # Verify the response
    assert response == "Hello World"
```

- This code can run without modifications against a Lambda function that is deployed. To do so, set the running_locally flag to False. This will set up the AWS SDK to connect to the AWS Lambda service in the cloud.

- **generate-event:** Generates mock serverless events. Using these, you can develop and test locally on functions that respond to asynchronous events such as those from Amazon S3, Kinesis, and DynamoDB. The following displays the command options available to the generate-event operation.

```
NAME:
   $sam local generate-event -
   Generates Lambda events (e.g. for S3/Kinesis etc) that can be piped to 'sam local invoke'

USAGE:
   $sam local generate-event command [command options] [arguments...]

COMMANDS:
     s3          Generates a sample Amazon S3 event
     sns         Generates a sample Amazon SNS event
     kinesis     Generates a sample Amazon Kinesis event
     dynamodb    Generates a sample Amazon DynamoDB event
     api         Generates a sample Amazon API Gateway event
     schedule    Generates a sample scheduled event

OPTIONS:
   --help, -h  show help
```

You can also use the sam local generate-event command to generate sample event payloads for 50+ events.

To make local development and testing of Lambda functions easier, you can generate and customize event payloads for the following services:

- Amazon Alexa
- Amazon API Gateway
- AWS Batch
- AWS CloudFormation
- Amazon CloudFront
- AWS CodeCommit
- AWS CodePipeline
- Amazon Cognito
- AWS Config
- Amazon DynamoDB
- Amazon Kinesis
- Amazon Lex
- Amazon Rekognition
- Amazon S3
- Amazon SES
- Amazon SNS
- Amazon SQS
- AWS Step Functions

- **validate:** Validates your template against the official AWS Serverless Application Model specification. The following is an example.
- ```
  $ sam validate
  ERROR: Resource "HelloWorld", property "Runtime": Invalid value node.
  Valid values are "nodejs4.3", "nodejs6.10", "nodejs8.10", "java8", "python2.7",
  "python3.6"(line: 11; col: 6)

  # Let's fix that error...
  $ sed -i 's/node/nodejs6.10/g' template.yaml

  $ sam validate
  Valid!
  ```
- **package** and **deploy**: sam package and sam deploy implicitly call AWS CloudFormation's package and deploy commands. For more information on packaging and deployment of SAM applications, see Packaging and Deployment.
- The following demonstrates how to use the package and deploy commands in SAM CLI.
- ```
  # Package SAM template
  $ sam package --template-file sam.yaml --s3-bucket mybucket --output-template-file packaged.yaml

  # Deploy packaged SAM template
  $ sam deploy --template-file ./packaged.yaml --stack-name mystack --capabilities CAPABILITY_IAM
  ```
- **sam logs:** sam logs command allows you you fetch logs generated by your Lambda function from the command line and includes several targeting and filtering options. For more information and samples, see Fetch, tail, and filter Lambda function logs.

The full documentation on SAM Local appears here.

IDE setup for SAM Local

VISUAL STUDIO

1. Using VS Code for debugging Node.js functions on AWS Lambda requires the following configuration under launch.json.

```
{
   "version": "0.2.0",
   "configurations": [
      {
         "name": "Attach to SAM Local",
         "type": "node",
         "request": "attach",
         "address": "localhost",
         "port": 5858,
         "localRoot": "${workspaceRoot}",
         "remoteRoot": "/var/task",
         "protocol": "legacy"
      }
   ]
}
```

2. Trigger your function using SAM Local CLI and choose the 'Attach to SAM Local' option.
3. That's it. You can debug your Node functions!

WEBSTORM

1. Trigger your function using the SAM Local command (See SAM Local section above).
2. Add an 'Attach to Node.js/Chrome' configuration with the same port used for the SAM Local function invocation.
3. Click debug and choose the configuration for SAM Local.
4. That's it! You are now able to debug your Node Lambda.

Serverless Offline Plugin

The Serverless Offline Plugin is an open-source tool that emulates both AWS Lambda and the API Gateway locally by running an HTTP server.

To get the plugin working:

1. Install the plugin:
   `$npm install serverless-offline --save-dev`

2. Add it to your plugins in your serverless.yml:
   ```
   plugins:
    -serverless-offline
   ```

3. In your project root run:
   `$serverless offline start` or
   `$sls offline start`.

The following, from Serverless Offline Plugin's GitHub, explains how to use its debugging features:

Debugging process:
Serverless offline plugin will respond to the overall framework settings and output additional information to the console in debug mode. In order to do this you will have to set the SLS_DEBUG environmental variable. You can run the following in the command line to switch to debug mode execution.

Unix: export SLS_DEBUG=*
Windows: SET SLS_DEBUG=*

Interactive debugging is also possible for your project if you have installed the node-inspector module and chrome browser. You can then run the following command line inside your project's root.

Initial installation: `$npm install -g node-inspector`
For each debug run: `$node-debug sls offline`

The system will start in wait status. This will also automatically start the chrome browser and wait for you to set breakpoints for inspection. Set the breakpoints as needed and, then, click the play button for the debugging to continue.

Depending on the breakpoint, you may need to call the URL path for your function in separate browser window for your serverless function to be run and made available for debugging.

Google Cloud Functions local emulator:

The Google Cloud Functions Node.js Emulator is a Node.js application that implements the Cloud Functions REST and gRPC APIs, and includes a command line interface for managing the application. The Emulator allows you to deploy, run, and debug your Cloud Functions on your local machine before deploying them to the production Cloud Functions service.

These instructions to use the local emulator are based on the official documentation:

1. $ npm install -g @google-cloud/functions-emulator

2. **Provide a Project ID as shown below before starting the Emulator:**

```
$ functions config set projectId [YOUR_PROJECT_ID]
```

3. **To invoke a function deployed to the Emulator, use the call command:**

```
$ functions call helloWorld
```

4. **To send a payload to the function, use the --data flag:**

```
$ functions call helloWorld --data='{"message":"Hello World"}'
```

5. **You can debug your functions running in the Emulator using either the standard Node.js Debugger or the new V8 Inspector integration. To debug using the standard Node.js Debugger, run the following command:**

```
$ functions debug helloWorld
```

6. **To debug using the new V8 Inspector integration, run the following command:**

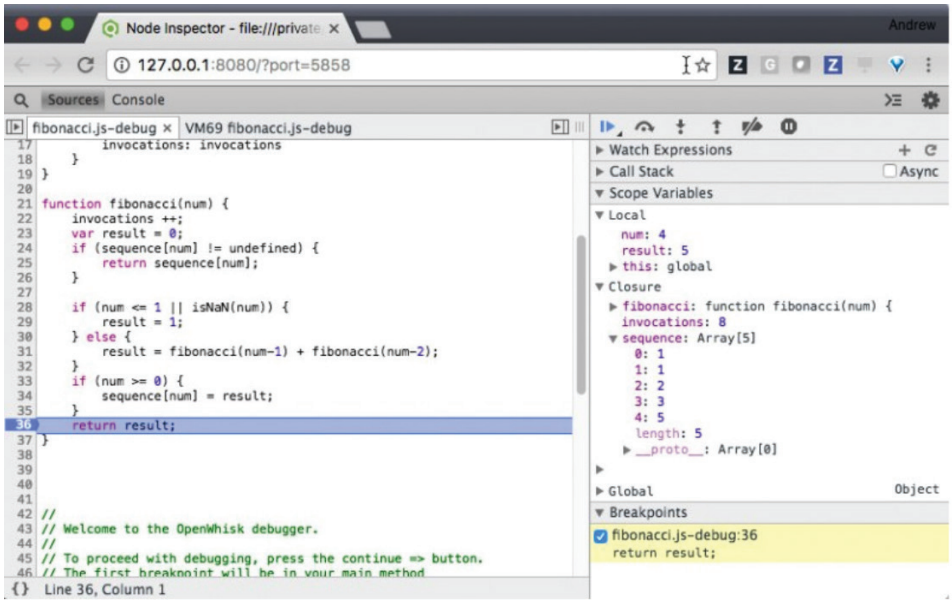
```
$ functions inspect helloWorld
```

wskdb ,the Openwhisk debugger

Wskdb "allows you to debug OpenWhisk actions on your local machine, in a rich debugging environment, just like you are debugging local code".

According to the author of the debugger, Nick Mitchell, the debugger currently requires Node 6 to 6.2.
Here we will go through the entire installation process:

1. $npm install -g openwhisk-debugger
2. Launch the debugger using the wskdb command
3. Run $attach <FUNCTION_NAME>' command
4. Run $invoke <FUNCTION_NAME> -p <PARAMETERS>'
5. This will launch the debug process.
6. For JavaScript/Node.js actions, it is modelled on the Chrome/Webkit/Node Inspector debugger:



See advanced options for wskdn commands:
https://www.ibm.com/blogs/bluemix/2017/01/using-new-openwhisk-debugger/

Azure Functions Core Tools

From Microsoft's documentation:

"Azure Functions Core Tools lets you develop and test your functions on your local computer from the command prompt or terminal. Your local functions can connect to live Azure services, and you can debug your functions on your local computer using the full Functions runtime. You can even deploy a function app to your Azure subscription."

To run a Functions project, run the Functions host. The host enables triggers for all functions in the project:
func host start
func host start supports the following options:

Option	Description
--port -p	The local port to listen on. Default value: 7071.
--debug <type>	Starts the host with the debug port open so that you can attach to the func.exe process from Visual Studio Code or Visual Studio 2017. The <type> options are VSCode and VS.
--cors	A comma-separated list of CORS origins, with no spaces.
	The port for the node debugger to use. Default: A value from launch.json or 5858
--debugLevel -d	The console trace level (off, verbose, info, warning, or error). Default: Info.
--timeout -t	The timeout for the Functions host to start, in seconds. Default: 20 seconds.
--useHttps	Bind to https://localhost:{port} rather than to http://localhost:{port}. By default, this option creates a trusted certificate on your computer.
--pause-on-error	Pause for additional input before exiting the process. Used when launching Core Tools from Visual Studio or VS Code.

When the Functions host starts, it outputs the URL of HTTP-triggered functions:

Found the following functions:
Host.Functions.MyHttpTrigger

Job host started
Http Function MyHttpTrigger: http://localhost:7071/api/MyHttpTrigger
For additional options, see this tutorial.

Production debugging:

ROOKOUT - HTTPS://WWW.ROOKOUT.COM/

Rookout provides a comprehensive solution for production debugging for all cloud architectures and specifically for serverless functions.

Rookout allows you to debug your functions running in production without actually breaking them. It basically collects all the data you need on the fly without needing to restart or redeploying your app, and with no need to write more code.

Rookout supports Node.js, JVM Languages, PyPy, Python 2.7 and Python 3.0. In addition, Rookout works with Serverless framework and Chalice. You can debug with Rookout on any desired cloud provider and even debug locally.

Using Rookout requires a token provided by the company. Request credentials here.

DEBUG NODEJS FOR OPENWHISK

1. Install the rookout dependency : `npm install --save rookout` and adding it in the entry file `const rookout = require('rookout/openwhisk');`
2. Update the package.json file to build dependencies on Linux using Docker:

```
"scripts" : {
    "install-modules": "docker run -it -v `pwd`:`pwd` -w `pwd` node:6 npm install",
    "build-package": "zip -r action.zip *",
    "build": "npm run install-modules && npm run build-package"
}
```

3. Connect the Rookout SDK on initialization:

```
const rookout = require('rookout/openwhisk');

rookout.connect('cloud.agent.rookout.com', 443, ORG_TOKEN);
```

4. Wrap your function with the OpenWhisk wrapper:

```
const rookout = require('rookout/openwhisk');

exports.main = rookout.wrap(myAction);
```

DEBUG PYTHON LAMBDAS

1. Install the rookout dependency : `pip install rook -t .` (installs module in root folder)
2. Wrap your function by using our Rookout lambda decorator like so:

```
from rook.serverless import serverless_rook

@serverless_rook
def lambda_handler(event, context):
    return 'Hello World'
```

3. Set Lambda environment for `ROOKOUT_AGENT_HOST` (cloud.agent.rookout.com), `ROOKOUT_AGENT_PORT` (443) and `ROOKOUT_TOKEN` in order to connect to a remote hosted agent

DEBUG NODEJS AWS LAMBDAS:

1. Install the rookout dependency : `npm install --save rookout` and add it in the entry file `const rookout = require('rookout/lambda');`

2. Wrap your function with the Lambda wrapper:
`const rookout = require('rookout/lambda');`

```
exports.handler = rookout.wrap((event, context, callback) => {
    callback(null, "Hello World");
});
```

3. Set Lambda environment for `ROOKOUT_AGENT_HOST` (cloud.agent.rookout.com), `ROOKOUT_AGENT_PORT` (443) and `ROOKOUT_TOKEN` in order to connect to a remote hosted agent

DEBUG CHALICE FUNCTIONS

1. Set the Rookout Agent host, port and token via environment variables in .chalice/config.json:

```
"environment_variables": {
    "ROOKOUT_AGENT_HOST": "cloud.agent.rookout.com",
    "ROOKOUT_AGENT_PORT": "443",
    "ROOKOUT_TOKEN": "<token>"
  }
```

2. Add `rook` to your `requirements.txt`

3. Initialize your Chalice app with `RookoutChalice`

```
from rook.serverless import RookoutChalice

app = RookoutChalice(app_name='python-aws-chalice')
```

You can use then the Chalice API as usual.

DEBUG SERVERLESS FRAMEWORK PYTHON

In order to deploy Rookout within your Serverless framework deployment tool follow these steps:

1. Install the rookout dependency : `pip install rook`

2. Wrap your function by using our Rookout lambda decorator like so:

```
from rook.serverless import serverless_rook

@serverless_rook
def lambda_handler(event, context):
    return 'Hello World'
```

3. $ sls plugin install -n serverless-python-requirements

4. Add to your serverless.yml the serverless-python-requirements plugin:

```
plugins:
 - serverless-python-requirements
```

1. Set Lambda environment for `ROOKOUT_AGENT_HOST` (cloud.agent.rookout.com), `ROOKOUT_AGENT_PORT` (443) and `ROOKOUT_TOKEN` in order to connect to a remote hosted agent.

```
environment:
 ROOKOUT_AGENT_HOST: cloud.agent.rookout.com
 ROOKOUT_AGENT_PORT: 443
 ROOKOUT_TOKEN:<YOUR_TOKEN>
```

1. Deploy your serverless functions using the serverless framework:

```
$ serverless deploy
```

2. Go to app.rookout.com and start debugging !

DEBUGGING JAVA LAMBDA WITH MAVEN

There are 4 simple steps to integrate Rookout into your existing Java application for an [agentless] setup:

1. Add maven dependencies to pom.xml file

```
<dependencies>
    <dependency>
                <groupId>com.amazonaws</groupId>
                <artifactId>aws-lambda-java-core</artifactId>
                <version>1.1.0</version>
    </dependency>
    <dependency>
                <groupId>com.rookout</groupId>
                <artifactId>rook</artifactId>
                <version>[0.1.28,)</version>
    </dependency>
```

```
        <dependency>
                <groupId>com.sun</groupId>
                <artifactId>tools</artifactId>
                <version>1.7.0.13</version>
        </dependency>
</dependencies>
```

IMPORTANT: the **com.sun.tools** is from the nuiton repository.
("http://maven.nuiton.org/release/")

Make sure you reference the following repository

```
<repositories>
        <repository>
        <id>nuiton</id>
        <name>nuiton</name>
        <url>http://maven.nuiton.org/release/</url>
        </repository>
</repositories>
```

2. Create an xml file in the project's root directory as a descriptor for the maven assembly plugin OR use a pre-defined descriptor

```
<assembly xmlns="http://maven.apache.org/ASSEMBLY/2.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/ASSEMBLY/2.0.0 http://maven.apache.org/xsd/assembly-2.0.0.xsd">
        <id>jar-with-dependencies</id>
        <formats>
                <format>jar</format>
        </formats>
        <includeBaseDirectory>false</includeBaseDirectory>
        <dependencySets>
                <dependencySet>
                <outputDirectory>/lib</outputDirectory>
                <useProjectArtifact>true</useProjectArtifact>
                <unpack>false</unpack>
                <scope>runtime</scope>
                <unpackOptions>
                        <excludes>
                        <exclude>META-INF/**</exclude>
                        </excludes>
                </unpackOptions>
                </dependencySet>
        </dependencySets>
</assembly>
```

3. Inside your main function, call LoadRook (add import com.rookout.rook.API;)

For example:

```
import com.rookout.rook.API;

public class TestLambda implements RequestHandler<Object, String> {
  @Override
  public String handleRequest(Object myCount, Context context) {

                    API.Load();

                    // Your awesome code here :)
  }
```

4. Set the following environment variables in your Lambda configuration as part of your deploying process

```
ROOKOUT_AGENT_HOST=cloud.agent.rookout.com
ROOKOUT_AGENT_PORT=443
ROOKOUT_TOKEN=<org_token>
```

DEBUGGING JAVA LAMBDA

There are 3 simple steps to integrate Rookout into your existing Java application for an [agentless] setup:

1. Add gradle dependencies
   `compile group: 'com.sun', name: 'tools', version: '1.7.0.13'`
   `compile group: 'com.rookout', name: 'rook', version: '0.1.12'`

IMPORTANT:: the com.sun.tools is from the nuiton repository.
("http://maven.nuiton.org/release/")

2. Inside your main function, call LoadRook (add import com.rookout.rook.API;)
   `API.Load();`

3. Set the Rook's agent configuration as environment variables in the Lambda configuration

Pipeline data to your desired destination
Rookout allows you to pipeline your data to every destination you want. You can send data to your current logging systems (Loggly, Splunk, ELK etc…) or pipeline it into Rookouts debugging IDE.

Full Rookout documentation.

About the author:

Zohar Einy is a Solutions Manager at Rookout - the rapid production debugging solution. Previously Zohar worked as a software developer for the IDF and as an engineer at TLV Partners Venture Capital. Zohar is enthusiastic about serverless and cloud infrastructure technologies.

Credits:

https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/
https://serverless.com/framework/docs/providers/aws/cli-reference/invoke-local/
https://serverless.com/framework/docs/providers/google/cli-reference/invoke-local/
https://serverless.com/framework/docs/providers/openwhisk/cli-reference/invoke-local/
https://hackernoon.com/running-and-debugging-aws-lambda-functions-locally-with-the-serverless-framework-and-vs-code-a254e2011010
https://aws.amazon.com/about-aws/whats-new/2018/04/aws-sam-cli-releases-new-init-command/
https://github.com/dherault/serverless-offline
https://github.com/GoogleCloudPlatform/cloud-functions-emulator
https://www.ibm.com/blogs/bluemix/2017/01/using-new-openwhisk-debugger/
https://docs.microsoft.com/en-us/azure/azure-functions/functions-run-local
Https://rookout.com
https://docs.rookout.com
https://docs.microsoft.com/en-us/azure/azure-functions/functions-run-local#passing-test-data-to-a-function