Debugging Lambda Functions Locally

This document will cover how to set up your local environment to run and debug lambda functions locally on your PC.

Setup

Before we get started, we will need to ensure a few things are installed:

- AWS CLI - https://docs.aws.amazon.com/cli/latest/userguide/install-cliv2.html
- SAM CLI - https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-cli-install.html
- Docker - https://docs.docker.com/get-docker/

Note: On Windows or OSX, you will need to install docker desktop. On Linux, follow the installation instructions for your distribution, and be jealous of the fancy desktop interface of your Windows and OSX brethren.

The AWS CLI will be used to grab a few values we will set as global params, such as DBUser and DBPassword. SAM CLI is what will build our code, and it uses Docker to run a container that will serve as the runtime.

Starting the API

After everything above is installed, open up a terminal and head to the lambda-functions project. From there, you should be able to run `sam build`. This will compile all the handlers specified in the `template.yml` file, and store them in a folder called `.aws-build`. The next step is to run the start-api command, which has quite a bit of configuration in it:

```
sam local start-api --parameter-overrides "DBUser=$value DBPassword=$value SSTAwsAccessKeyId=$value SSTAwsSecretAccessKey=$value"
```

In the command above, I specify a number of overrides to use for out local development. This is because the template in this project uses dynamic references, something that is not supported by `sam local`. These values can be fetched using the AWS CLI with the following commands:

- `aws secretsmanager get-secret-value --secret-id dev-mysql-access`
- `aws secretsmanager get-secret-value --secret-id dev-aws-access-keys`

The output of these commands will contain the values to use above. If you require some assistance locating them or running these commands, just give someone a poke and they will help you out. In the future, it would be possible for our lambdas to dynamically fetch these values so that we do not need to pass them all in, but this is our solution for now.

Once you fill all the values in and run the command, you should be met with something that looks like this:

```
You can now browse to the above endpoints to invoke your functions. You do not need to restart/reload SAM CLI whi
le working on your functions, changes will be reflected instantly/automatically. You only need to restart SAM CLI
 if you update your AWS SAM template
2021-02-26 13:08:45  * Running on http://127.0.0.1:3000/ (Press CTRL+C to quit)
```

This means that everything is up and running, and you should be able to use Postman (or any other tool to make API requests) and hit `localhost:3000`. A couple things to keep in mind:

- When hitting localhost, you do not need to include `/v1/` in the URL
- A valid `Authorization` header is still required for a protected route, but it will not expire.

Like the command line states, you do not need to restart this if you make changes to code, though you do need to run `sam build` again. Therefore, it might be beneficial to have this running in a separate process that can stay open while you do development

One last thing to note, please ensure the version of Lambda-Dependencies being imported is at least version 1.0.6. Older versions will have trouble interfacing with various AWS services (S3, Cognito, etc).

Debugging the API

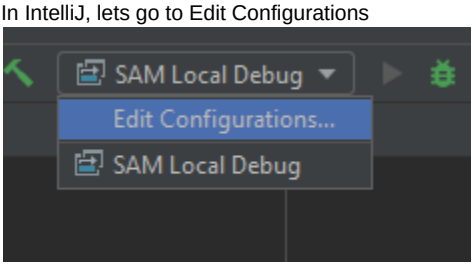
Debugging the API is done by running the start-api command with the -d flag. This will specify that we want to run in debug mode, and open a port to attach a debugger to the remote JVM. In this example we will set this up with IntelliJ, though the concept should be applicable to any IDE that supports remote JVM Debugging.

Lets start by running the above command with some modifications:

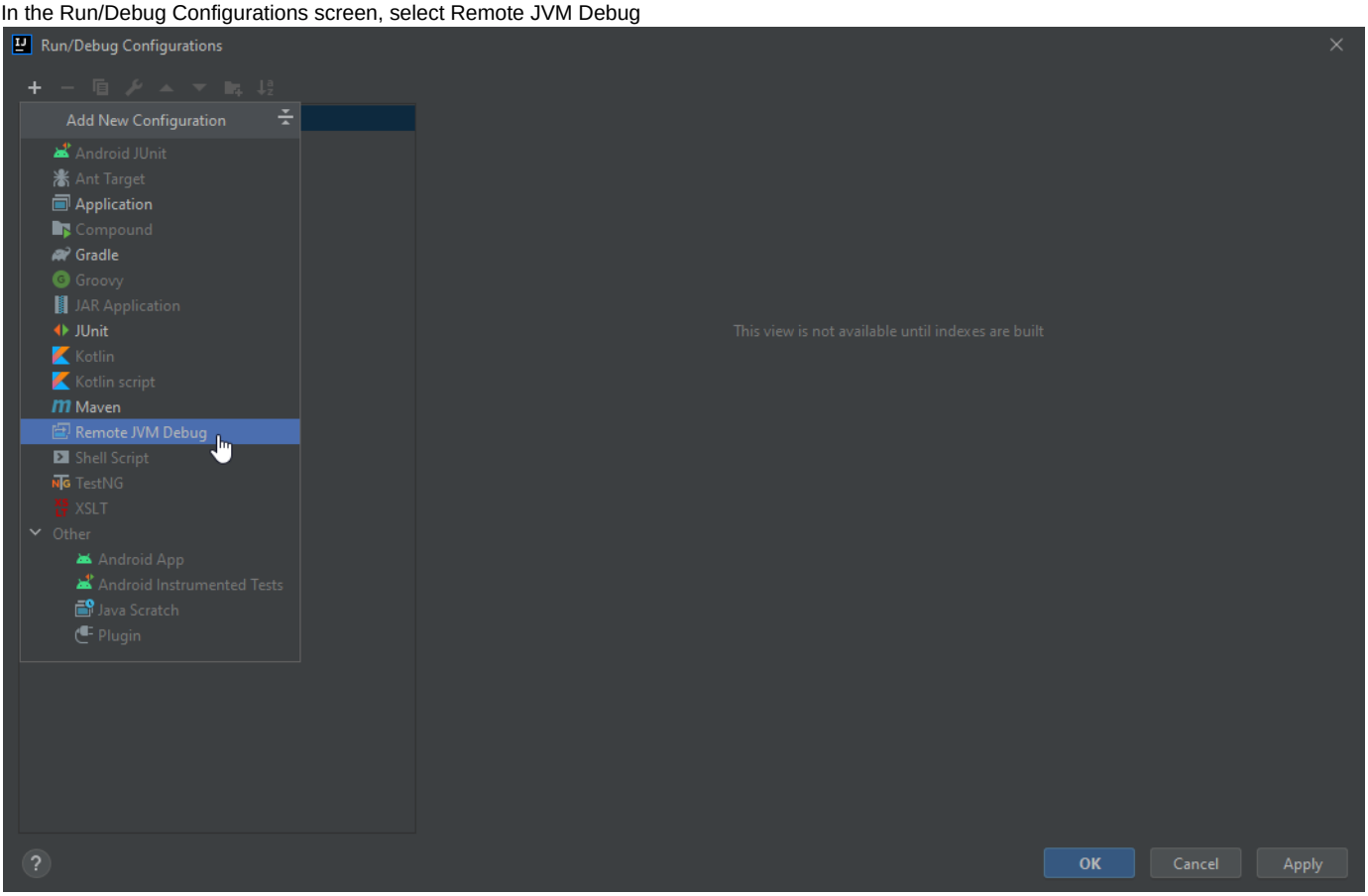
```
sam local start-api -d 5858 --parameter-overrides "{your-overrides}"
```

The port we specify doesn't really matter, it just needs to match what we set up in out intelliJ config.

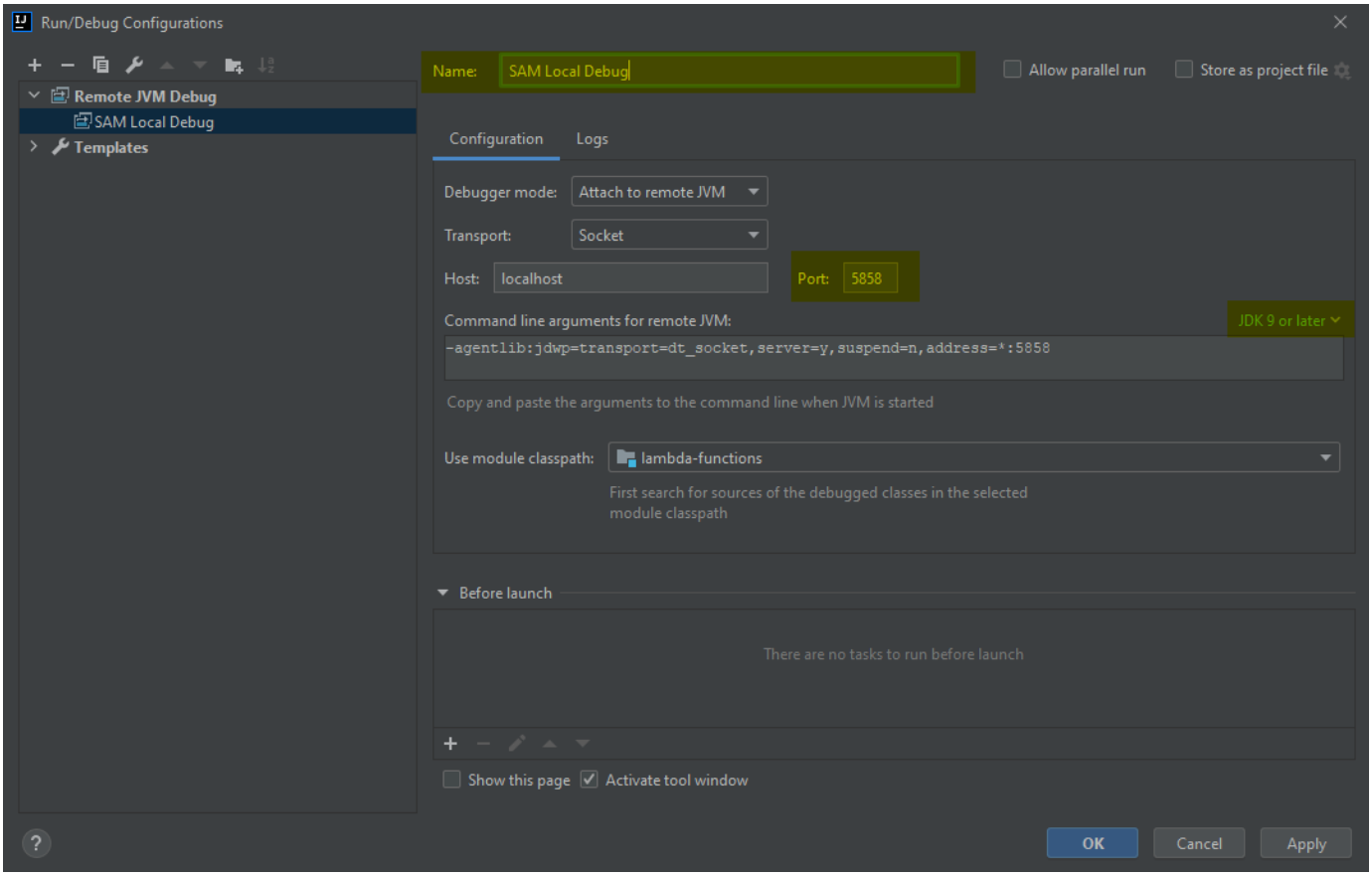
In IntelliJ, lets go to Edit Configurations



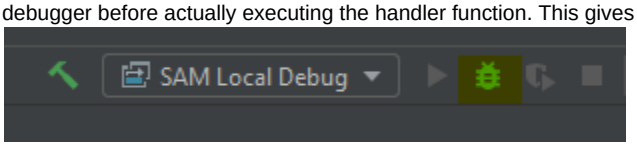
In the Run/Debug Configurations screen, select Remote JVM Debug



Next, Set the JDK verion, match the port to what we specified, and provide a config name. Then hit OK



That's all the setup that we need to do in IntelliJ. Now, if we use a tool like Postman to make a request, the function will wait for us to attach a debugger before actually executing the handler function. This gives us the opportunity to set our breakpoints, and then hit the debug icon.



Once they attach, you should be able to step through your code like you normally would and examine any values etc.