

Using LocalStack to simulate AWS services

Remember when we said that the lack of a proper local development environment is a bit of a pain when developing serverless applications? To fix that, instead of deploying our code to AWS every time we want to test it, we can use LocalStack to run it locally. LocalStack allows us to mimic the functionality of AWS services, such as DynamoDB and S3, on our own machine. This way, we can test and develop our cloud and serverless apps offline.

Installing LocalStack

To use LocalStack, we first need to install its **command-line interface (CLI)**. The following is a detailed set of instructions:

1. To be able to install the CLI, you need to have the following installed on your computer:
 - Python (3.7 up to 3.10)
 - Pip (Python package manager)
 - Docker
2. Once you have those set up, you can proceed to your terminal and run the following command:

```
$ python3 -m pip install localstack
```

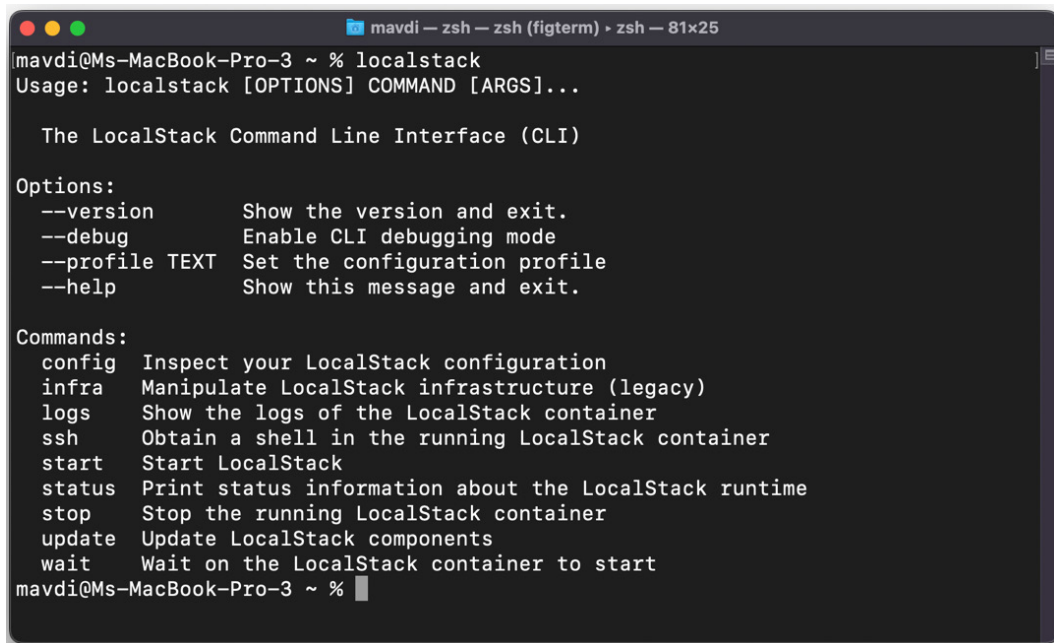
3. If everything goes well, you should see this prompt upon the command's completion:

```
Successfully installed localstack-1.3.1
```

4. To confirm that LocalStack has been installed, you can run this command in your terminal:

```
$ localstack --help
```

5. If you see the following prompt, you're good to go:

A terminal window titled 'mavdi — zsh — zsh (figterm) • zsh — 81x25'. The prompt is 'mavdi@Ms-MacBook-Pro-3 ~ %'. The user has entered 'localstack'. The output shows the usage and options for the LocalStack CLI. The options are: --version (Show the version and exit.), --debug (Enable CLI debugging mode), --profile TEXT (Set the configuration profile), and --help (Show this message and exit.). The commands are: config (Inspect your LocalStack configuration), infra (Manipulate LocalStack infrastructure (legacy)), logs (Show the logs of the LocalStack container), ssh (Obtain a shell in the running LocalStack container), start (Start LocalStack), status (Print status information about the LocalStack runtime), stop (Stop the running LocalStack container), update (Update LocalStack components), and wait (Wait on the LocalStack container to start). The prompt is now 'mavdi@Ms-MacBook-Pro-3 ~ %' with a cursor.

```
mavdi@Ms-MacBook-Pro-3 ~ % localstack
Usage: localstack [OPTIONS] COMMAND [ARGS]...

The LocalStack Command Line Interface (CLI)

Options:
  --version      Show the version and exit.
  --debug        Enable CLI debugging mode
  --profile TEXT Set the configuration profile
  --help         Show this message and exit.

Commands:
  config  Inspect your LocalStack configuration
  infra   Manipulate LocalStack infrastructure (legacy)
  logs    Show the logs of the LocalStack container
  ssh     Obtain a shell in the running LocalStack container
  start   Start LocalStack
  status  Print status information about the LocalStack runtime
  stop    Stop the running LocalStack container
  update  Update LocalStack components
  wait    Wait on the LocalStack container to start
mavdi@Ms-MacBook-Pro-3 ~ %
```

Figure 8.1 – LocalStack command output

Note

In some Linux environments, by default, `PATH` will not recognize the location where Pip installs its packages. So, you need to manually add that to `PATH` by adding `export PATH=$HOME ./local/bin:$PATH` to your `.bashrc` or `.zshrc` file.

Starting LocalStack

Now that it's installed, we can start LocalStack by running this command:

```
$ localstack start
```

Once it's finished running, you should see this prompt with `Ready .` at the end:

```

> localstack start

LocalStack CLI 1.3.1

[18:37:59] starting LocalStack in Docker mode
LocalStack Runtime Log (press CTRL-C to quit)
Waiting for all LocalStack services to be ready
2023-01-11 21:38:00,026 CRIT Supervisor is running as root. Privileges were not dropped because no user is specified in the config file. If you intend to run as root, you can set user=root in the config file to avoid this message.
2023-01-11 21:38:00,027 INFO supervisord started with pid 16
2023-01-11 21:38:01,030 INFO spawned: 'infra' with pid 21
2023-01-11 21:38:02,031 INFO success: infra entered RUNNING state, process has stayed up for > than 1 seconds (startsecs)

LocalStack version: 1.3.2.dev
LocalStack Docker container id: 095a5f6a6656
LocalStack build date: 2023-01-10
LocalStack build git hash: d5bb7f19

2023-01-11T21:38:02.277 WARN --- [-functhread3] hypercorn.error : ASGI Framework Lifespan error, continuing without Lifespan support
2023-01-11T21:38:02.277 WARN --- [-functhread3] hypercorn.error : ASGI Framework Lifespan error, continuing without Lifespan support
2023-01-11T21:38:02.279 INFO --- [-functhread3] hypercorn.error : Running on https://0.0.0.0:4566 (CTRL + C to quit)
2023-01-11T21:38:02.279 INFO --- [-functhread3] hypercorn.error : Running on https://0.0.0.0:4566 (CTRL + C to quit)
Ready.

```

Figure 8.2 – Expected LocalStack command output

You can check whether LocalStack was started correctly by running the `docker container ls` command in another terminal window. You should see the LocalStack container listed:

```

> docker container ls
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS
PORTS
NAMES
856788c1ff30   localstack/localstack              "docker-entrypoint.sh"   21 seconds ago Up 20 seconds
(healthy)     127.0.0.1:4510-4559->4510-4559/tcp, 127.0.0.1:4566->4566/tcp, 127.0.0.1:4571->4571/tcp, 127.0.0.1:12121->12121/tcp, 5678/tcp   localstack_main

```

Figure 8.3 – Docker command listing all the existing containers

Note

To learn about alternative methods for installing LocalStack, check out their documentation website: <https://docs.localstack.cloud/getting-started/installation/>.

With that taken care of, our next step is to install `cdklocal`. This is a CLI wrapper required to deploy the CDK code to LocalStack using its APIs. This can be done by installing it globally, as an npm library, by running the following command:

```
$ npm install -g aws-cdk-local
```

To ensure it was installed properly, you can run this command:

```
$ cdklocal --version
```

Afterward, you should see a prompt that looks like the following:

```
> cdklocal --version  
2.59.0 (build b24095d)
```

Figure 8.4 – Current cdklocal version

Now, with LocalStack and cdklocal installed and running, navigate to the project's infrastructure folder. We are going to put some logic in the `lib/chapter-8-stack.ts` file that, if using LocalStack, will deploy only DynamoDB.

Your code will look like the following:

```
export class Chapter8Stack extends Stack {  
  public readonly acm: ACM;  
  
  public readonly route53: Route53;  
  
  public readonly s3: S3;  
  
  public readonly vpc: Vpc;  
  
  public readonly dynamo: DynamoDB;  
  
  constructor(scope: Construct, id: string, props?: StackProps) {  
    super(scope, id, props);  
  
    const isCDKLocal = process.env.NODE_ENV === 'CDKLocal';  
  
    this.dynamo = new DynamoDB(this, `Dynamo-${process.env.NODE_ENV ||  
    ''}`);  
  
    if (isCDKLocal) return;  
  
    this.route53 = new Route53(this, `Route53-${process.env.NODE_ENV ||  
    ''}`);  
  
    this.acm = new ACM(this, `ACM-${process.env.NODE_ENV || ''}`, {  
      hosted_zone: this.route53.hosted_zone,  
    });  
  
    this.s3 = new S3(this, `S3-${process.env.NODE_ENV || ''}`, {
```

```
        acm: this.acm,
        route53: this.route53,
    });

    new ApiGateway(this, `Api-Gateway-${process.env.NODE_ENV || ''}`,
    {
        route53: this.route53,
        acm: this.acm,
        dynamoTable: this.dynamo.table,
    });
}
```

In the terminal, within the `infrastructure` folder, run the following command:

```
$ yarn cdklocal bootstrap
```

Next, run the following command:

```
$ yarn cdklocal deploy
```

The first command will bootstrap the environment on your local cloud, while the second command will deploy the stack into the same local cloud. The output of the second command should look similar to this:



```
✔ Chapter8Stack-Development
🌟 Deployment time: 5.41s

Stack ARN:
arn:aws:cloudformation:us-east-1:000000000000:stack/Chapter8Stack-Development/da2092cb

🌟 Total time: 10.04s

Done in 11.51s.
```

Figure 8.5 – Message displayed after a stack was successfully deployed

Congratulations! You just deployed the development stack into your local cloud infrastructure.

Note

Keep in mind that LocalStack is ephemeral, meaning if you run the `localstack stop` command, any stack that was deployed previously will be destroyed.

Configuring DynamoDB to work with LocalStack

By default, the LocalStack endpoint is on port 4588 on your localhost. To make sure DynamoDB is reaching the LocalStack endpoint, rather than the default AWS one, we need to configure its endpoint:

1. First, go to `infrastructure/lib/constructs/Lambda/post/lambda/index.ts` where you can find the following code:

```
const tableName = process.env.TABLE_NAME!
const awsRegion = process.env.REGION || 'us-east-1';

const dynamoDB = new DynamoDB.DocumentClient({
  region: awsRegion,
  endpoint:
    Process.env.DYNAMODB_ENDPOINT || `https://
dynamodb.${awsRegion}.amazonaws.com`,
});
```

Here, we are passing a custom endpoint to the `DocumentClient()` DynamoDB API from a `.env` file that we'll create later. Notice the inline logic in this property; this value will only be present in the `.env` file when running the local server. When deploying the stack to AWS, the default URL will be passed to the `DocumentClient()` DynamoDB API.

2. Now, navigate to `/infrastructure/lib/constructs/Lambda/get/lambda/index.ts` and make the same changes to `DocumentClient()` there:

```
const dynamoDB = new DynamoDB.DocumentClient({
  region: awsRegion,
  endpoint:
    process.env.DYNAMODB_ENDPOINT || `https://
dynamodb.${awsRegion}.amazonaws.com`,
});
```

3. The final step is to create an environment file containing all the necessary information to run our local server.

Go to the `server/` folder and create a `.env` file at the root of the folder with the following variables and values:

```
PORT=3000
REGION=us-east-1
TABLE_NAME=todolist-cdklocal
DYNAMODB_ENDPOINT=http://localhost:4566
```

In the terminal, inside the `server` folder, run the following command:

```
$ export AWS_PROFILE=cdk
```

Although it is not mandatory to include your real access key ID and secret access key, LocalStack needs a string value in those properties to simulate the AWS calls. Instead of adding these values to the `.env` file, you can export your AWS profile as we did in previous chapters. Otherwise, LocalStack will produce an error saying `Missing credentials in config`.

4. Still in the `server/` folder, in your terminal, run the following command:

```
$ yarn dev
```

5. Once the command is executed, the local development server will start and you should see a message similar to the following, indicating that the server is running:

```
> yarn dev
yarn run v1.22.19
$ tsnd src/index.ts
[INFO] 18:40:05 ts-node-dev ver. 2.0.0 (using ts-node ver. 10.8.2, typescript ver. 4.8.4)
Server is listening on port 3000
█
```

Figure 8.6 – Message indicating the local server is operational on a specified port

6. You can then test the server by making requests to `localhost:3000`, for example, by sending a request to the `/healthcheck` path to check whether the server is working properly:

```
> curl --request GET \
  --url http://localhost:3000/healthcheck
"OK"█
```

Figure 8.7 – Health check path response

7. Another example is hitting the `POST /` endpoint to create a table item:

```
> curl --request POST \
  --url http://localhost:3000/ \
  --header 'Content-Type: application/json' \
  --data '{
    "todo": {
      "todo_name": "Test Item",
      "todo_description": "Test description",
      "todo_completed": true
    }
  }'
{"todo":{"id":"3c9860da-fef7-4c46-b6d8-d83c227628ef","todo_completed":true,"todo_description":"Test description","todo_name":"Test Item"}}█
```

Figure 8.8 – POST root path response

8. You can also hit the GET / endpoint to retrieve all items from the table:

```
> curl --request GET \
  --url http://localhost:3000/
{"todos":[{"todo_name":"Test Item","id":"3c9860da-fef7-4c46-b6d8-d83c227628ef","todo_description":"Test description","todo_completed":true}]}
```

Figure 8.9 – GET root path response

Great – we are all set. Our serverless application is running locally and we have massively reduced the amount of time it takes for us to see the result of our changes since we ran the entire stack locally.

Limitations of LocalStack

We dislike it when a solution to a certain problem is proposed without covering the limitations of that solution. All of that is left for the developers to find out themselves (looking at you, AWS docs) through trial and error, potentially wasting hours or days on a solution that might not be the right one for them.

We’ve presented LocalStack as a silver bullet for local serverless development, but it too comes with limitations that you might find annoying or impossible to work with. In our company, we use LocalStack on some projects and not on others, mainly depending on the AWS services we are using.

One of the main limitations is that LocalStack may not fully replicate the behavior of some AWS services. For example, the behavior of the S3 service in LocalStack may differ from that of AWS’s actual S3 service, which can lead to unexpected errors or behavior when the code is deployed to production. Additionally, not all AWS services are supported by LocalStack, so certain CDK constructs may not be able to be fully tested locally. This is also evident when working with certain versions of AWS Lambda containers for Node.js. Right now, only Node.js v14.x.x is supported by LocalStack.

AWS’s official solution for the local development of CDK applications is to use the official **AWS SAM** toolkit. AWS SAM is an open source framework for building serverless applications. It provides a simplified way of defining the Amazon API Gateway APIs, AWS Lambda functions, and Amazon DynamoDB tables needed by your serverless application. By using AWS SAM, you can define your application’s resources using YAML or JSON (ugh, so 2008) templates, which are easier to read and maintain than traditional CloudFormation templates.

The SAM CLI toolkit can be used to read the configurations of a Lambda function and have it invoked locally as such:

```
# Invoke MyFunction from the TODOSTack
sam local invoke -t ./cdk.out/TODOSTack.template.json MyFunction
```

You can find out more about this method of local testing by following this URL: <https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-cdk-testing.html>.

Ultimately, while the tooling for local serverless development might not be mature, many efforts are being made right now to close the gaps. We recommend keeping an eye on the latest tooling to evolve the way you code serverless infrastructure as the space becomes more mature.

Summary

In this chapter, we discussed how to streamline serverless development by using a local express server and LocalStack for a local cloud environment. We provided step-by-step instructions for installing and running LocalStack and `cdklocal`, configuring the DynamoDB endpoint to point to LocalStack, and importing the same function used in the deployed Lambda function to the local server. Additionally, we showed how to run commands and make requests to the local development server to test the code's functionality. The overall objective of this process is to enable the local testing and development of AWS cloud infrastructure without incurring costs or impacting production resources. In the next chapter, we will introduce the concept of **indestructible serverless application architecture (ISAA)**, a design pattern for building serverless applications that are highly resilient and can scale infinitely.