Localstack for AWS — White Paper

Author: Sivadoss Ram

Localstack – An Introduction

Localstack is a cloud service emulator that runs in a single container on your laptop or in your CI environment. With LocalStack, you can run your AWS applications or Lambdas entirely on your local machine without connecting to a remote cloud provider!

Whether you are testing complex CDK applications or Terraform configurations, or just beginning to learn about AWS services, LocalStack helps speed up and simplify your testing and development workflow.

LocalStack supports a growing number of AWS Services, like AWS Lambda, S3, DynamoDB, Kinesis, SQS, SNS and more.

Even the pro version support more additional APIs and advance features, the community version supports 80+ AWS services.

Localstack CLI Installation

Linux/Windows:
You can also install the LocalStack CLI directly in your PYthon environment. Please make sure to install the following tools on your machine before moving ahead:

- Python (Python 3.7 up to 3.11 is supported) - https://docs.python.org/3/using/index.html
- Pip (Python package manager) - https://pip.pypa.io/en/stable/installation/

Afterwards you can install the LocalStack CLI in your Python environment with:

python3 -m pip install --upgrade localstack

Start LocalStack inside a Docker container by running:

```
$ localstack start -d

    __                     _____ __             __
   / /   ____  _____ _/ / ___// /_____ _____/ /__
  / /   / __ \/ ___/ __ `/ /\__ \/ __/ __ `/ ___/ //_/
 / /___/ /_/ / /__/ /_/ / /___/ / /_/ /_/ / /__/ ,<
/_____/\____/\___/\__,_/_//____/\__/\__,_/\___/_/|_|

    💻 LocalStack CLI 3.4.0
    👤 Profile: default

[12:47:13] starting LocalStack in Docker mode 🐳
localstack.py:494
           preparing environment
bootstrap.py:1240
           configuring container
bootstrap.py:1248
           starting container
bootstrap.py:1258
[12:47:15] detaching
bootstrap.py:1262
```

You can query the status of respective services on LocalStack by running:

```
% localstack status services
```

Service	Status	
acm	✓ available	
apigateway	✓ available	
cloudformation	✓ available	

```
| cloudwatch            | ✔ available |
| config                | ✔ available |
| dynamodb              | ✔ available |
...
```

Besides using the CLI, there are other ways of starting and managing your LocalStack instance:

- LocalStack Desktop
  Get a desktop experience and work with your local LocalStack instance via the UI.
- LocalStack Docker Extension
  Use the LocalStack extension for Docker Desktop to work with your LocalStack instance.
- Docker-Compose
  Use docker-compose to configure and start your LocalStack Docker container.
- Docker
  Use the docker CLI to manually start the LocalStack Docker container.
- Helm
  Use helm to create a LocalStack deployment in a Kubernetes cluster.

LocalStack runs inside a Docker container, and the above options are different ways to start and manage the LocalStack Docker container.

You can simply start Docker container by executing docker run command for community and pro versions as follows.

```
docker run \
  --rm -it \
  -p 127.0.0.1:4566:4566 \
  -p 127.0.0.1:4510-4559:4510-4559 \
  -v /var/run/docker.sock:/var/run/docker.sock \
  localstack/localstack
```

```
docker run \
  --rm -it \
  -p 127.0.0.1:4566:4566 \
  -p 127.0.0.1:4510-4559:4510-4559 \
  -p 127.0.0.1:443:443 \
  -e LOCALSTACK_AUTH_TOKEN=${LOCALSTACK_AUTH_TOKEN:?} \
  -v /var/run/docker.sock:/var/run/docker.sock \
  localstack/localstack-pro
```

If you want to deploy LocalStack in your Kubernetes cluster, you can use Helm.

Prerequisites

- Kubernetes (Kubernetes.io)
- Helm (https://helm.sh/docs/intro/install/)

Deploy LocalStack using Helm

You can deploy LocalStack in a Kubernetes cluster by running these commands:

```
helm repo add localstack-repo https://helm.localstack.cloud
helm upgrade --install localstack localstack-repo/localstack
```

How to Update

The LocalStack CLI allows you to easily update the different components of LocalStack. To check the various options available for updating, run:

```
localstack update --help
Usage: localstack update [OPTIONS] COMMAND [ARGS]...

  Update different LocalStack components.

Options:
  -h, --help  Show this message and exit.

Commands:
  all  -  Update all LocalStack components
  docker-images  -  Update docker images LocalStack depends on
  localstack-cli  -  Update LocalStack CLI
```

Updating the LocalStack CLI itself (localstack update localstack-cli and localstack update all) is currently only supported if you installed the CLI in a Python environment.

Configure Backend

The backend url for localstack is **http://localstack:4566**

After starting localstack container, check if all the services are running by hitting the backend url on browser: http://localhost:4566/health

```
{
  - services: {
        cloudformation: "running",
        cloudwatch: "running",
        dynamodb: "running",
        dynamodbstreams: "running",
        firehose: "running",
        iam: "running",
        sts: "running",
        kinesis: "running",
        lambda: "running",
        logs: "running",
        s3: "running",
        ses: "running",
        sns: "running",
        sqs: "running",
        apigateway: "running"
  },
  - features: {
        persistence: "initialized",
        initScripts: "initialized"
  }
}
```
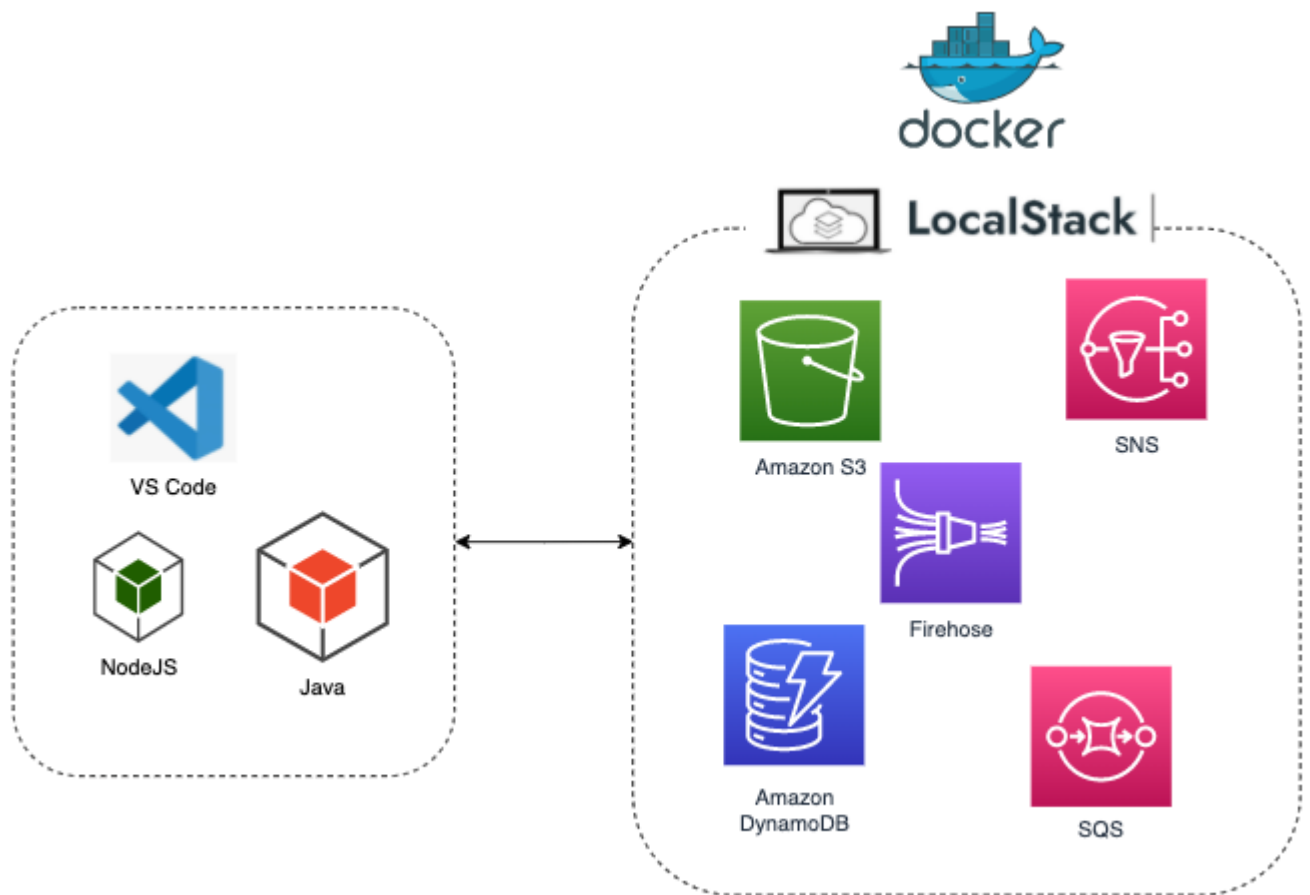
How to configure IDE vscode for Terraform/Boto3 with localstack

```
aws configure
...
AWS Access Key ID [****************VKHB]:
AWS Secret Access Key [****************gbOt]:
Default region name [ap-south-1]:
Default output format [json]:

cat ~/.aws/credentials
[default]
aws_access_key_id = test
aws_secret_access_key = test
region = us-west-2
output = json
```

Terraform configuration for localstack

Terraform is an Infrastructure-as-Code (IaC) framework developed by HashiCorp. It enables users to define and provision infrastructure using a high-level configuration language. Terraform uses HashiCorp Configuration Language (HCL) as its configuration syntax. HCL is a domain-specific language designed for writing configurations that define infrastructure elements and their relationships.

LocalStack supports Terraform via the AWS provider through custom service endpoints as following.

AWS provider: https://registry.terraform.io/providers/hashicorp/aws/latest/docs

Custom service endpoint: https://registry.terraform.io/providers/hashicorp/aws/latest/docs/guides/custom-service-endpoints#localstack

You can configure Terraform to use LocalStack in two ways:

- Using the tflocal wrapper script (https://github.com/localstack/terraform-local) to automatically configure the service endpoints for you.
- Manually configuring the service endpoints in your Terraform configuration with additional maintenance.

TfLocal the wrapper script

tflocal is a small wrapper script to run Terraform against LocalStack. tflocal script uses the terraform override mechanism and creates a temporary file localstack_providers_override.tf to configure the endpoints for the AWS provider section. The endpoints for all services are configured to point to the LocalStack API (http://localhost:4566 by default). It allows you to easily deploy your unmodified Terraform scripts against LocalStack.

Create a Terraform configuration

Create a new file named main.tf and add a minimal S3 bucket configuration to it. The following contents should be added in the main.tf file:

```
resource "aws_s3_bucket" "test-bucket" {
  bucket = "my-bucket"
}
```

Install the tflocal wrapper script

To install the tflocal command, you can use pip (assuming you have a local Python installation):

```
pip install terraform-local
```

After installation, you can use the tflocal command, which has the same interface as the terraform command line.

```
tflocal --help
<disable-copy>
Usage: terraform [global options] <subcommand> [args]
...
<disable-copy>
```

Manual Configuration

Instead of using the tflocal script, you have the option to manually configure the local service endpoints and credentials. The following sections will provide detailed steps for this manual configuration.

General Configuration

To begin, you need to define mock credentials for the AWS provider. Specify the following in your main.tf file:

```
provider "aws" {

  access_key = "test"
  secret_key = "test"
  region     = "ap-south-1"
}
```

Request Management

Next, to prevent routing and authentication issues (which are unnecessary in this context), you should provide some general parameters:

```
provider "aws" {

  access_key                 = "test"
  secret_key                 = "test"
  region                     = "ap-south-1"


  # only required for non-virtual hosted-style endpoint use case.
  #
https://registry.terraform.io/providers/hashicorp/aws/latest/docs#s3_use_pa
th_style
  s3_use_path_style          = true
  skip_credentials_validation = true
  skip_metadata_api_check    = true
  skip_requesting_account_id = true
}
```

Services

Furthermore, it's necessary to configure the individual services to use LocalStack. For S3, this configuration resembles the following snippet, where we've chosen to use the virtual hosted-style endpoint:

```
  endpoints {
    s3             = "http://s3.localhost.localstack.cloud:4566"
  }
```

Final Configuration

The final minimal configuration for deploying an S3 bucket via a main.tf file should resemble the following:

```
provider "aws" {
```

```
    access_key                  = "mock_access_key"
    secret_key                  = "mock_secret_key"
    region                      = "us-east-1"

    s3_use_path_style           = true
    skip_credentials_validation = true
    skip_metadata_api_check     = true
    skip_requesting_account_id  = true

    endpoints {
      s3             = "http://s3.localhost.localstack.cloud:4566"
    }
}

resource "aws_s3_bucket" "test-bucket" {
  bucket = "my-bucket"
}
```

Endpoint Configuration

Here's a configuration example with additional service endpoints. Please note that these provider configurations may not be necessary if you use the `tflocal` script (as described above). You can save the following configuration in a file named `provider.tf` and include it in your Terraform configuration.

```
provider "aws" {
  access_key                  = "test"
  secret_key                  = "test"
  region                      = "us-east-1"
  s3_use_path_style           = false
  skip_credentials_validation = true
  skip_metadata_api_check     = true
  skip_requesting_account_id  = true

  endpoints {
    apigateway     = "http://localhost:4566"
    apigatewayv2   = "http://localhost:4566"
    cloudformation = "http://localhost:4566"
    cloudwatch     = "http://localhost:4566"
    dynamodb       = "http://localhost:4566"
    ec2            = "http://localhost:4566"
    es             = "http://localhost:4566"
    elasticache    = "http://localhost:4566"
    firehose       = "http://localhost:4566"
    iam            = "http://localhost:4566"
    kinesis        = "http://localhost:4566"
    lambda         = "http://localhost:4566"
    rds            = "http://localhost:4566"
    redshift       = "http://localhost:4566"
    route53        = "http://localhost:4566"
    s3             = "http://s3.localhost.localstack.cloud:4566"
    secretsmanager = "http://localhost:4566"
    ses            = "http://localhost:4566"
    sns            = "http://localhost:4566"
    sqs            = "http://localhost:4566"
```

```
    ssm             = "http://localhost:4566"
    stepfunctions   = "http://localhost:4566"
    sts             = "http://localhost:4566"
  }
}
```

To heuristically detect whether your Terraform configuration should be deployed against LocalStack, you can use the following snippet:

```
data "aws_caller_identity" "current" {}
output "is_localstack" {
  value = data.aws_caller_identity.current.id == "000000000000"
}
```

It will detect whether the AWS account ID is 000000000000, which is the default value for LocalStack. If you use a different account ID within LocalStack, you can customize the snippet accordingly.

How to configure Boto3?

This is an easy-to-use Python client for localstack (https://github.com/localstack/localstack) The client library provides a thin wrapper around boto3 (https://github.com/boto/boto3) which automatically configures the target endpoints to use LocalStack for your local cloud application development.

To make use of this library, you need to have LocalStack installed on your local machine. In particular, the localstack command needs to be available.

Installation

The easiest way to install *LocalStack* is via pip:

```
pip install localstack-client
```

Usage

This library provides an API that is identical to boto3's. A minimal way to try it out is to replace import boto3 with import localstack_client.session as boto3. This will allow your boto3 calls to work as normal.

For example, to list all s3 buckets in localstack:

```
import localstack_client.session as boto3
client = boto3.client('s3')
response = client.list_buckets()
```

Another example below shows using localstack_client directly. To list the SQS queues in your local (LocalStack) environment, use the following code:

```
import localstack_client.session
```

```
session = localstack_client.session.Session()
sqs = session.client('sqs')
assert sqs.list_queues() is not None
```

If you use `boto3.client` directly in your code, you can mock it.

```
import localstack_client.session
import pytest

@pytest.fixture(autouse=True)
def boto3_localstack_patch(monkeypatch):
    session_ls = localstack_client.session.Session()
    monkeypatch.setattr(boto3, "client", session_ls.client)
    monkeypatch.setattr(boto3, "resource", session_ls.resource)
sqs = boto3.client('sqs')
assert sqs.list_queues() is not None  # list SQS in localstack
```

We can use the following environment variables for configuration:

- `AWS_ENDPOINT_URL`: The endpoint URL to connect to (takes precedence over `USE_SSL`/`LOCALSTACK_HOST` below)
- `LOCALSTACK_HOST` (deprecated): A `<hostname>:<port>` variable defining where to find LocalStack (default: `localhost:4566`).
- `USE_SSL` (deprecated): Whether to use SSL when connecting to LocalStack (default: `False`).

Enabling Transparent Local Endpoints

The library contains a small `enable_local_endpoints()` util function that can be used to transparently run all `boto3` requests against the local endpoints.

The following sample illustrates how it can be used - after calling `enable_local_endpoints()`, the S3 `ListBuckets` call will be run against LocalStack, even though we're using the default boto3 module.

```
import boto3
from localstack_client.patch import enable_local_endpoints()
enable_local_endpoints()
# the call below will automatically target the LocalStack endpoints
buckets = boto3.client("s3").list_buckets()
```

The patch can also be unapplied by calling `disable_local_endpoints()`:

```
from localstack_client.patch import disable_local_endpoints()
disable_local_endpoints()
# the call below will target the real AWS cloud again
buckets = boto3.client("s3").list_buckets()
```

What is CDKTF ?

Cloud Development Kit for Terraform (CDKTF) allows you to use familiar programming languages to define cloud infrastructure and provision it through HashiCorp Terraform.

CDKTF leverages existing libraries and tools to **help convert the definitions you write in your preferred programming language to Terraform configuration files instead of the Terraform HCL language.** This enables developers to leverage their existing programming skills and the ecosystem of their language of choice when developing infrastructure components.

Configuration

To configure your existing CDKTF configuration to work with LocalStack, manually configure the local service endpoints and credentials. It includes:

- General configuration to specify mock credentials for the AWS provider (region, access_key, secret_key).
- Request Management to avoid issues with routing and authentication, if needed.
- Service configuration to point the individual services to LocalStack.

Here is a configuration example to use with Python & TypeScript CDKTF configurations:

Python: localstack_config.py

```
AWS_CONFIG = {
  "region": "us-east-1",
  "endpoints": [
      {
          "apigateway": "http://localhost:4566",
          "apigatewayv2": "http://localhost:4566",
          "cloudformation": "http://localhost:4566",
          "cloudwatch": "http://localhost:4566",
          "dynamodb": "http://localhost:4566",
          "ec2": "http://localhost:4566",
          "es": "http://localhost:4566",
          "elasticache": "http://localhost:4566",
          "firehose": "http://localhost:4566",
          "iam": "http://localhost:4566",
          "kinesis": "http://localhost:4566",
          "lambda": "http://localhost:4566",
          "rds": "http://localhost:4566",
          "redshift": "http://localhost:4566",
          "route53": "http://localhost:4566",
          "s3": "http://s3.localhost.localstack.cloud:4566",
          "secretsmanager": "http://localhost:4566",
          "ses": "http://localhost:4566",
          "sns": "http://localhost:4566",
          "sqs": "http://localhost:4566",
          "ssm": "http://localhost:4566",
          "stepfunctions": "http://localhost:4566",
          "sts": "http://localhost:4566",
      }
```

```
        ],
}
```

TypeScript: localstack-config.ts

```typescript
export const AWS_CONFIG = {
  region: "us-east-1",
  endpoints: [
    {
      apigateway: "http://localhost:4566",
      apigatewayv2: "http://localhost:4566",
      cloudformation: "http://localhost:4566",
      cloudwatch: "http://localhost:4566",
      dynamodb: "http://localhost:4566",
      ec2: "http://localhost:4566",
      es: "http://localhost:4566",
      elasticache: "http://localhost:4566",
      firehose: "http://localhost:4566",
      iam: "http://localhost:4566",
      kinesis: "http://localhost:4566",
      lambda: "http://localhost:4566",
      rds: "http://localhost:4566",
      redshift: "http://localhost:4566",
      route53: "http://localhost:4566",
      s3: "http://s3.localhost.localstack.cloud:4566",
      secretsmanager: "http://localhost:4566",
      ses: "http://localhost:4566",
      sns: "http://localhost:4566",
      sqs: "http://localhost:4566",
      ssm: "http://localhost:4566",
      stepfunctions: "http://localhost:4566",
      sts: "http://localhost:4566",
    },
  ],
};
```

To get started with CDKTF on LocalStack, we will set up a simple stack to create some AWS resources. We will then deploy the stack to LocalStack, and verify that the resources have been created successfully. Before we start, make sure you have the following prerequisites:

- LocalStack
- Cdktf (https://www.npmjs.com/package/cdktf)

For Python:

- Python (https://www.python.org/downloads/)
- Pipenv (https://pipenv.pypa.io/en/latest/installation.html#installing-pipenv)

For TypeScript:

- Tsc (https://www.npmjs.com/package/typescript)

Create a new directory named cdktf-localstack and initialize a new CDKTF project using the following command:

```
$ cdktf init
...
? Do you want to continue with Terraform Cloud remote state management? No
? What template do you want to use? python

Initializing a project using the python template.
? Project Name sample-app
? Project Description A simple getting started project for cdktf.
? Do you want to start from an existing Terraform project? No
? Do you want to send crash reports to the CDKTF team? Refer to
https://developer.hashicorp.com/terraform/cdktf/create-and-
deploy/configuration-file#enable-crash-reporting-for-the-cli for more
information no
Note: You can always add providers using 'cdktf provider add' later on
? What providers do you want to use? aws
...
```

(Optional) If necessary, we can install the AWS provider separately for CDKTF, by running the following command:

Python:

$ pipenv install cdktf-cdktf-provider-aws

TypeScript:

$ npm install @cdktf/provider-aws

Add the following code to import the AWS provider and create a new S3 bucket in the relevant file:

Python: main.py

```python
#!/usr/bin/env python
from constructs import Construct
from cdktf import App, TerraformStack
from cdktf_cdktf_provider_aws.provider import AwsProvider
from cdktf_cdktf_provider_aws.s3_bucket import S3Bucket


class MyStack(TerraformStack):
    def __init__(self, scope: Construct, id: str):
```

```python
        super().__init__(scope, id)

        AwsProvider(self, "aws",
            region="us-east-1",
            s3_use_path_style=True,
            endpoints=[
                {
                    "s3": "http://localhost:4566",
                    "sts": "http://localhost:4566",
                }
            ]
        )

        S3Bucket(self, "bucket")

app = App()
MyStack(app, "cdktf-example-python")

app.synth()
```

TypeScript: main.ts

```typescript
import { Construct } from "constructs";
import { App, TerraformStack } from "cdktf";
import { AwsProvider } from "@cdktf/provider-aws/lib/provider";
import {S3Bucket} from "@cdktf/provider-aws/lib/s3-bucket";

class MyStack extends TerraformStack {
  constructor(scope: Construct, id: string) {
    super(scope, id);

    new AwsProvider(this, "aws",{
      region: "us-east-1",
      s3UsePathStyle: true,
      endpoints: [
          {
            s3: "http://localhost:4566",
            sts: "http://localhost:4566"
          },
      ],
    });

    new S3Bucket(this, "bucket", {});
  }
}

const app = new App();
new MyStack(app, "example");
app.synth();
```

Run the following command to compile and deploy the CDKTF stack to LocalStack:

```
$ cdktf synth && cdktf deploy
```

You should see the following output:
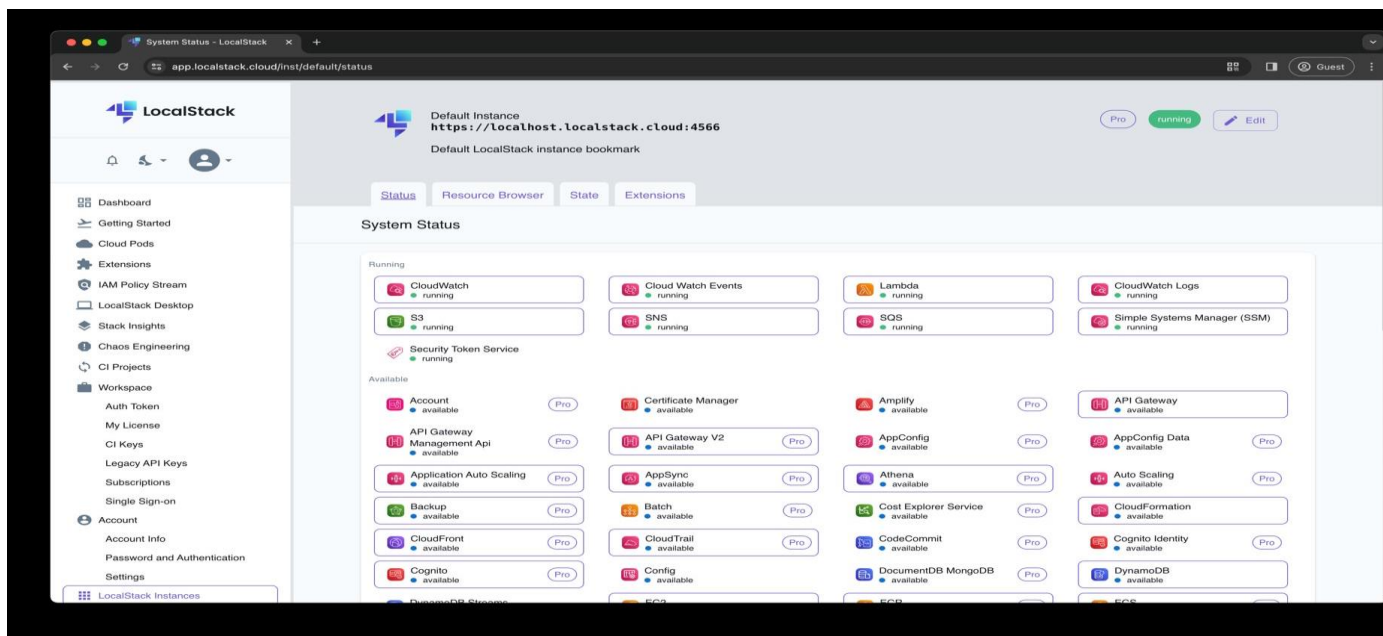
```
example  Initializing the backend...
example
        Successfully configured the backend "local"! Terraform will
automatically
        use this backend unless the backend configuration changes.
...
example  aws_s3_bucket.bucket (bucket): Creating...
example  aws_s3_bucket.bucket (bucket): Creation complete after 5s
[id=terraform-20230418074657926600000001]
example
        Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Verify that the S3 bucket has been created successfully by running the following command:

```
awslocal s3 ls
```

Your CDKTF stack is now successfully deployed to LocalStack. You can now start using CDKTF to create and manage your AWS resources on LocalStack

Resource Browser Console: https://localhost.localstack.cloud:4566

Key References

- ▶ https://localstack.cloud
- ▶ https://docs.localstack.cloud
- ▶ https://docs.localstack.cloud/getting-started/installation/
- ▶ https://pypi.org/project/localstack/
- ▶ https://docs.docker.com/desktop/install/windows-install/
- ▶ http://localhost:4566/health
- ▶ https://app.localstack.cloud/dashboard
- ▶ https://localhost.localstack.cloud:4566

Conclusion

- ▪ Use LocalStack as a drop-in replacement for AWS in your dev and testing environments
- ▪ Faster deployment of AWS infrastructure locally and on CI
- ▪ Develop and test your AWS applications locally to reduce development time and increase product velocity
- ▪ Allow your developers to focus their time and effort where it counts by developing and testing locally. Nobody likes setting up sandbox accounts, cleaning up resources or constantly monitoring AWS dev spending
- ▪ More than just an Emulator - By emulating cloud services locally, LocalStack enables additional features and workflows that are not feasible on the cloud.

"Our engineering team utilizes LocalStack to provide a complete, localized AWS environment where developers can build, test, profile and debug infrastructure and code ahead of deployment to the cloud."

Rick Timmis
Head of Engineering at Xiatech