Skripted.

Build a 100% Serverless 3-Tier Infrastructure Using CloudFormation.

Utilizing Infrastructure as Code and CI/CD to accelerate the development of web applications.

Maxim Schram
January 2023

Whitepaper

Includes ready-to-deploy CloudFormation scripts.



Table of contents

Introduction **4**

Chapter 1. Defining requirements **5**
- Use Case 5
- The Shopify Ecosystem 5
- Shopify app Requirements 5
- 100% Serverless 6
- CI/CD integration 6
- Zero trust 7

Chapter 2. The infrastructure in a nutshell **8**
- Web tier 8
- Application tier 8
- Database tier 9
- CI/CD using GitHub Workflows and ECR 9

Chapter 3. Technical deep dive **10**
- 2.1 CloudFormation stacks 11
- 2.2 IAM Role for GitHub 11
- 2.3 VPC 12
- 2.4 The Database tier 14
  - Secrets Manager, TLS and IAM authentication 15
  - Aurora Serverless V2 15
  - RDS Proxy 15
  - DynamoDB 16
  - Conclusion 16
- 2.5 The Application tier 17
  - Cognito User Pool and User Pool Client 17
  - Lambda API function 18
  - API Gateway 19
  - Conclusion 20
- 2.6 The Web tier 21
  - Web Secrets 22
  - App Runner 22
  - Synthetic Canary 23
- 2.7 Review 24

Appendix: Using the pre-built CloudFormation scripts **25**
Pre-requisites 26
Step by step 26

Introduction

DevSecOps has become essential to modern web development. However, infrastructure provisioning and management is often delegated as a side task, resulting in slower development cycles and increased security risks.

The aim of this whitepaper is to provide development teams with a quick start and solid baseline for building a reliable and scalable cloud infrastructure to support their web applications. Furthermore, the provided example utilizes serverless services and deployment automation to help reduce costs and accelerate development cycles.

The proposed infrastructure has been iteratively improved and tested in a real-world setting as part of a live Shopify application. However, its use case can be applied to many other web applications that consist of a public frontend with a private backend.

To get started, simply deploy the CloudFormation stacks available via GitHub, and use it as the foundation for your own setup.

If you need help at any point, feel free to drop me a message at max@skripted.io.

Happy provisioning!

Maxim Schram
Founder @ Skripted | Infrastructure as Code
www.skripted.io - max@skripted.io

Github repo: https://github.com/skripted-io/serverless-3-tier-cloudformation

Chapter 1. Defining requirements

Before taking a look at the proposed infrastructure, let's determine its requirements.

Use Case

The infrastructure presented in this whitepaper has been deployed in a real-world setting as part of a Shopify application. The application has a customer-facing frontend and a private API and database as the backend. Because this setup is common to many web applications, I will use it as an example throughout the remainder of this whitepaper.

The Shopify Ecosystem

Shopify provides a great ecosystem for merchants seeking to extend the functionality of their out-of-the-box online retail system. The ecosystem consists of thousands of apps built by a few thousand independent app developers.

Shopify offers all the tools developers need to deeply integrate their app with Shopify, such as APIs, SDKs, and ready-to-use templates in various languages like NodeJS and Ruby. However, there is one crucial thing that it does not provide: hosting. This can be highly discouraging for new Shopify developers because building a fault-tolerant, secure infrastructure can be a daunting task. And while AWS does provide tools to help developers get started quickly, like Elastic Beanstalk for example, such solutions generally don't provide the fine-grained controls that many applications need.

Being a Shopify developer who had to go through this process myself, who's also a certified AWS cloud architect, I decided to make the blueprint of my infrastructure available as a series of CloudFormation stacks. This hopefully saves you a lot of time.

Shopify app Requirements

Because we are building cloud infrastructure for a Shopify app, it is worthwhile to evaluate the requirements a typical Shopify application has.

- Most Shopify apps come with a relatively lightweight frontend built on languages such as REACT, Node.js, or Go.
- Then, (in most cases) the majority of processing is done at the API level.
- Most apps will need a database to store things people interact with.
- Because we are working with SDK and API keys, we need a way to safely store secrets.
- Finally, Shopify apps typically require fast user session storage.

The aforementioned requirements will benefit from a three-tier design so we can separate application components for security purposes. Then, because many Shopify shops experience seasonal spikes in traffic, it’s important our infrastructure is scalable. Lastly, outages will mean merchants are losing money, so, for that reason, our infrastructure needs to be highly available.

It's worth noting that even if you're developing for a different platform, many of these requirements probably still apply in one way or the other. Feel free to modify the design to fit your needs..

100% Serverless

In addition to building a robust architecture that meets the requirements, my goal also was to build it entirely using serverless services. That way, developers won't have to worry about server maintenance. Scaling for traffic spikes will be more manageable. And, the infrastructure will become more "financially-elastic", meaning you will pay less when traffic is low.

CI/CD integration

Automated infrastructure provisioning
Another goal was to make infrastructure provisioning part of the app development process. Typically, you will need at least a staging and production version of your app. As a best practice, you don't want those environments to share resources.

Also, you probably won't need a highly-available version of your infrastructure in your staging environment. And in some cases, each developer might even want to have their own hosted development environment.

For each of those use cases, CloudFormation is up to the task. You can describe your infrastructure in code and add variables to provision slightly modified versions per environment. GitHub Workflows can help you with automatic provisioning.

Automated code deployment
A similar approach can be applied to application deployment. Like with most modern development environments, you want to deploy quickly and often. This means you want your staging environment to always be in sync with the latest development changes, so it can be tested, updated and then deployed to production.

The easiest way to achieve this is using a containerized approach. Once the dev team merges their changes and deploys them to a staging branch on GitHub, a docker

container can be built automatically. The container is then pushed to a container registry on AWS (ECR), which triggers an update to the application..

Zero trust

Finally, my goal was to apply the principle of Zero Trust where possible. Not only because we will be working with customer data, but because in this day and age this should be the default.

Within AWS, a Zero Trust approach includes, but is not limited to, applying the Principle of Least Privilege, storing secrets in Secret Managers, using IAM authentication where possible, and preventing data from traversing the internet when not necessary.

Chapter 2. The infrastructure in a nutshell

The following is a simplified infrastructure diagram that takes into account most of what has been discussed so far.

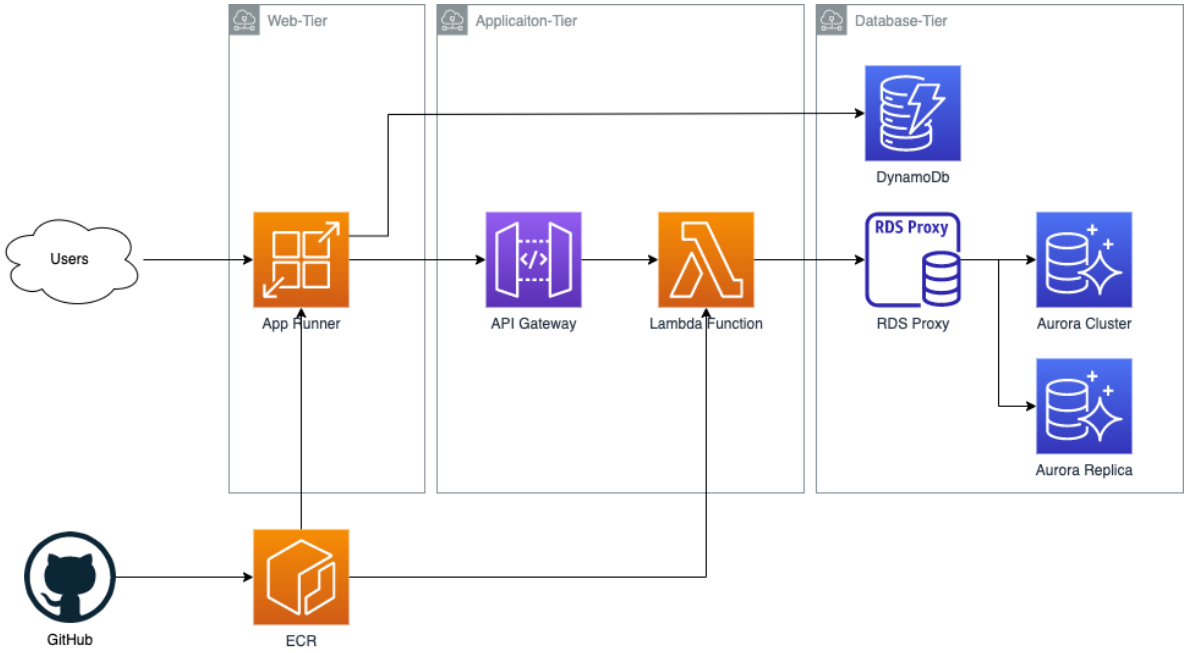


Fig 1. The architecture consists of a web tier, application tier and database tier.

Web tier

The web tier is where end-users (merchants and customers) interact with our Shopify app. In my case I'm using AWS App Runner to host the main application. AppRunner is a PaaS (Platform as a Service) solution. It deploys a highly available, scalable, and secure application server that runs in an isolated virtual private cloud (VPC), fully managed by AWS. It takes a code container and then deploys it automatically.

Application tier

The application tier is an API backend layer that takes instructions from App Runner and relays them to the database that runs in the database tier. It uses Amazon API Gateway with a private endpoint to receive the instructions which are then "proxied" to a Lambda function. The Lambda function queries the database and sends the response back to API Gateway, which in turn passes the response back to App Runner. A proxy server is used to manage the database connections.

Database tier

The database tier is where merchant, customer and session data will be stored.

SQL
Many Shopify apps will benefit from a relational database and in general, MySQL and PostgreSQL are good choices. Aurora supports those databases and also comes as a serverless version. Therefore, we will be using an Aurora Serverless V2 cluster, which is a quickly scalable Multi-AZ solution. We'll be putting an RDS Proxy in front of our cluster to prevent overloading it with Lambda connections (more on that later).

NoSQL
At the time of writing, Shopify apps need some form of session storage and Shopify itself is recommending Redis. While it would be an option to deploy an Elasticache Redis service, currently that type of solution does not come serverless. For this use case, however, DynamoDB provides a perfect alternative.

— I rewrote Shopify's Redis module to work with DynamoDB. Let me know if you're interested in using it.

Technically, DynamoDB does not run in a subnet because it's a public service. However, to be able to use it privately, we will be deploying a VPC Gateway Endpoint and adding a route to our private route table.

CI/CD using GitHub Workflows and ECR

Finally, as you can see in the diagram, GitHub is used to push code containers to ECR. Both AppRunner and Lambda are then configured to update whenever there is a new version available for them in ECR. This fulfills our continuous integration and continuous delivery needs.

Need help implementing this infrastructure?

E-mail me at max@skripted.io or
give me a call +1 (646) 543-2059.

Chapter 3. Technical deep dive

Below is the architecture shown in full. In this chapter I will look closer at each individual component, starting with the VPC and then working “upwards” from the database tier to the application tier and web tier.

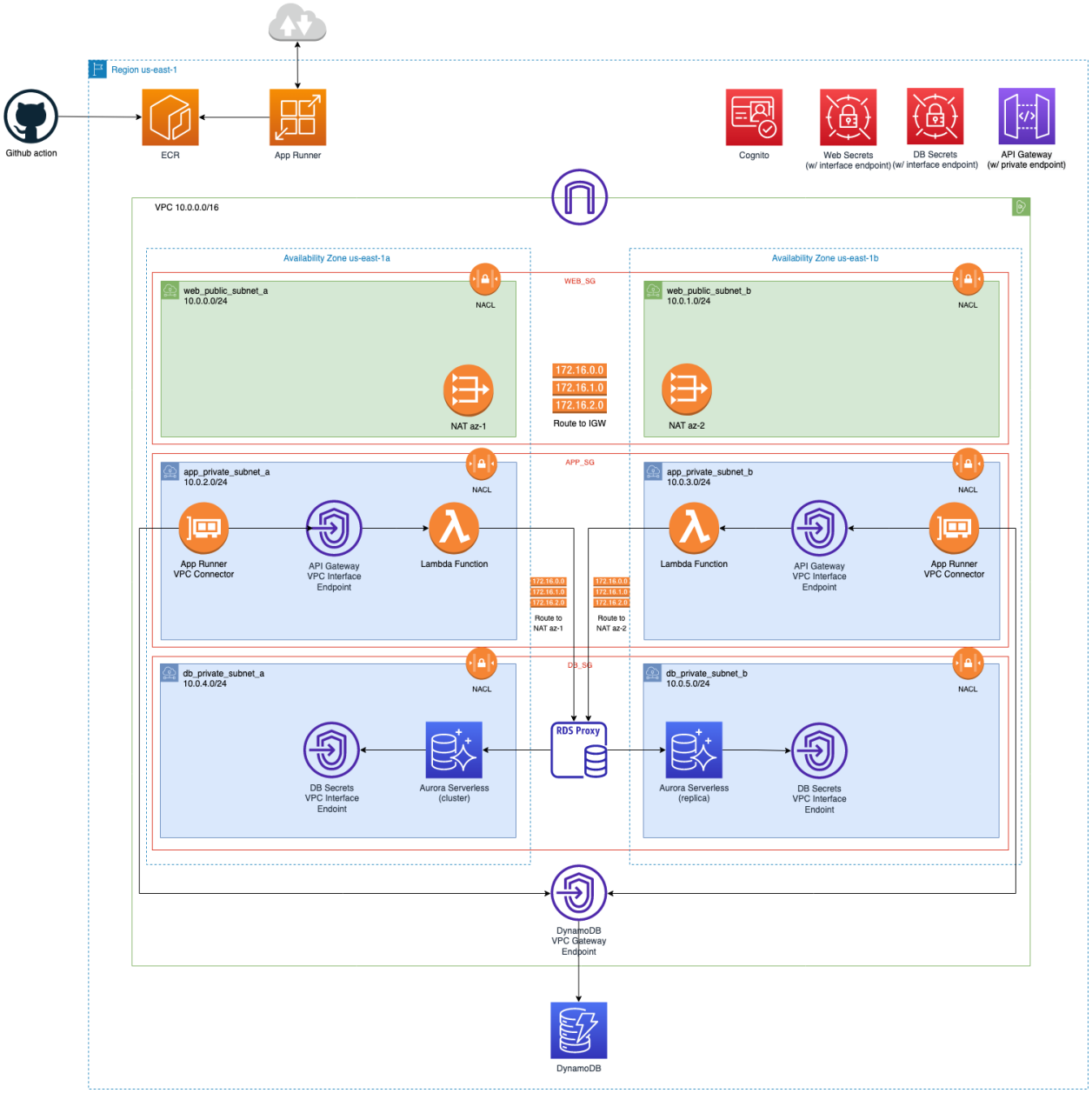


Fig 2. The architecture in full.

The CloudFormation scripts discussed in this whitepaper can be found on GitHub here. Feel free to download or fork the repository and make changes for your particular situation. Please keep in mind that in many cases the individual scripts reference resources created by one of the other scripts. So you'll have to make the changes accordingly.

2.1 CloudFormation stacks

In this whitepaper, each central service element comes with its own ready to deploy CloudFormation stack available as a script written in the YAML format.

The bundle of scripts can be found on my GitHub page and are structured as shown on the right. What follows is a summary of what is being deployed with every individual file and why.

To keep the whitepaper clean, I am not including the actual CloudFormation code which can sometimes be lengthy. Instead I reference the filename so you can keep GitHub to the side of it.

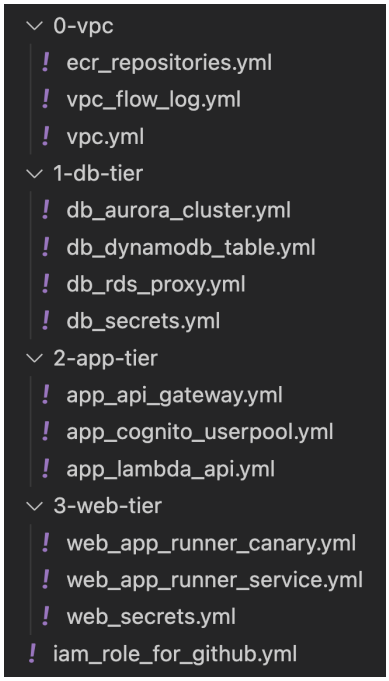


Fig 3. GitHub file structure.

2.2 IAM Role for GitHub

Filename: /iam_role_for_github.yml

While the entire CloudFormation stack is designed to be deployed automatically using a GitHub workflow, it will need permissions to be able to do so. I advise you to create a fresh AWS account, log in as an IAM user with full admin permissions. Then run this CloudFormation script manually.

What it will do is the following:

1. Create a role for GitHub to assume, so it can deploy our CloudFormation scripts.
2. Create an IAM entity to describe an identity provider (IdP) that supports OpenID Connect (OIDC). This is what GitHub will use to log in to your AWS account.
3. Create all the permissions that are needed to create our resources and attach it to the role in the first step.

Now you are ready to set up your GitHub workflow. See the appendix on how to do this.

2.3 VPC

`Filename: /0-vpc/vpc.yml`

The VPC is pretty straightforward with 6 subnets, an internet gateway, 2 NAT gateways, security groups and NACLs. Let's take a closer look.

Subnets
The VPC with a CIDR block of 10.0.0.0/16 consists of two public subnets (10.0.0.0/24 and 10.0.1.0/24) for the web tier, two private subnets (10.0.2.0/24 and 10.0.3.0/24) for the app tier tier, and two private subnets (10.0.4.0/24 and 10.0.5.0/24) for the database tier. Having two subnets for each tier will help us achieve high-availability which we will later see.

NACLs
Each subnet comes with a Network Access Control List (NACL) attached to it, allowing all egress and ingress traffic. The NACLs provide an extra layer of security in case we need it, such as blocking IP addresses in case of unauthorized access. However, in its initial configuration, all traffic is allowed.

Security Groups
Each subnet comes with a security group that allows traffic from itself on specific ports. This way resources in the subnet can communicate freely with one another. To enable Lambda in the app tier to communicate with the RDS Proxy in the db tier, two separate security groups are created for RDS Proxy and Lambda, allowing RDS Proxy ingress from Lambda on port 5432.

NAT Gateways
Two NAT gateways are deployed for high availability with a route from the private route table. Later, you will observe that we will attach App Runner to our private app subnet. Doing this will block App Runner from egress internet access and these NAT gateways will restore that traffic. In the case of our Shopify app, this is essential to access Shopify APIs.

VPC Flow Log

`Filename: /0-vpc/vpc_flow_log.yml`

To monitor traffic entering and exiting our VPC, we are setting up flow logs. This allows us to detect unauthorized access at any time. This stack creates a flow log group, a role for the Flow Log service to assume, and the necessary policies for that role to create and write logs.

ECR Repositories

`Filename: /0-vpc/ecr_repositories.yml`

White not technically part of the VPC, this seemed like the best place to deploy two ECR repositories. One to host our Lambda API function and one for our web application. Because I like to keep my repositories organized, I added a lifecycle policy to automatically expire (delete) untagged images after 24 hours.

2.4 The Database tier

The backend of our app will be an API gateway and a Lambda function which will query a database. Therefore, our database tier needs to accommodate for that. This is what we are building:

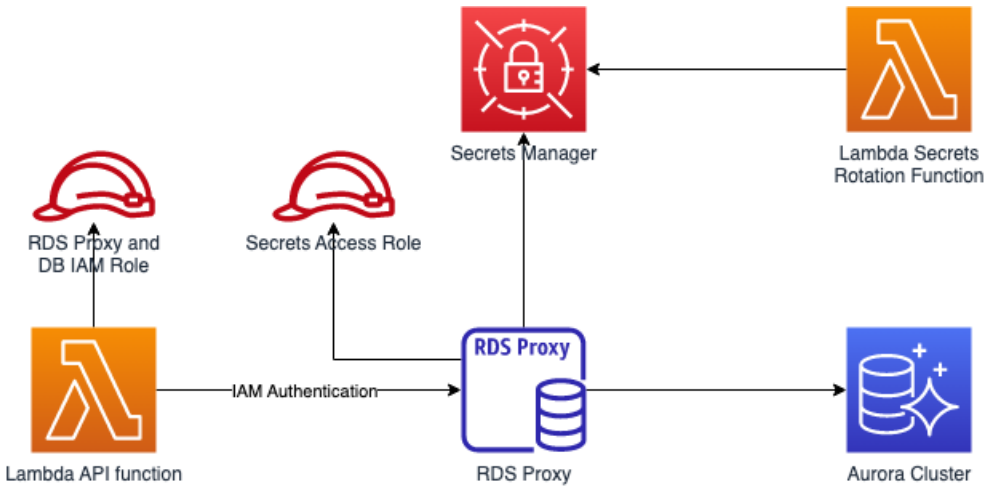


Fig 4. Logical structure of the database tier.

The Lambda function (from our app tier) queries an Amazon Aurora cluster via an RDS Proxy service. To do this Lambda will assume a role allowing it access to RDS Proxy and Aurora. To set this all up, these are the steps we will need to follow:

1. First we create a Secrets Manager secret. Contrary to its name, a “secret” can store multiple key/value combinations. Such as a database name, username and password. CloudFormation can generate an initial password when it creates the secret. Once created, our secret hosts the data we need to be able to create a database.
2. We create the database cluster and pass the credentials from our secret.
3. The missing link is the endpoint of our cluster which will be added to our secret by the Aurora cluster once its creation is complete. We deploy a separate Lambda function to rotate the DB password periodically.
4. Next we create the RDS Proxy with IAM authentication and TLS enabled and give it a role to access the secrets.
5. Finally we deploy our Lambda API function and give it a role to allow access to our RDS proxy using IAM authentication.

Let’s go build this!

Secrets Manager, TLS and IAM authentication
`Filename: /1-db-tier/db_secrets.yml`

Creation of the Secret is pretty straightforward. We pass the resource a secret string containing the name of the database and a username. In the script, you can customize them by modifying the variables at the beginning of the file. We ask the resource to generate a password and add this to the secret. For now this is all there is to it. But as you will later see we'll use it as a placeholder to store more values later.

Aurora Serverless V2
`Filename: /1-db-tier/db_aurora_cluster.yml`

Aurora clusters work with subnet groups, so we first define in which subnet we would like to deploy our cluster. In this case, the database subnets.

Next, we create the cluster. My Shopify app uses PostgreSQL and port 5543; however, feel free to change this to MySQL and/or another port of choice. We pass the database name, username, and password stored in Secrets Manager and assign the correct security group.

Next, we launch a writer instance in AZ1 and a reader instance in AZ2. The reader instance will be automatically promoted to a writer instance in case AZ1 fails.

The "SecretTargetAttachment" statement makes sure the endpoint address of our cluster is added to our Secrets Manager on completion so RDS Proxy will know where to connect to.

Finally, we use a CloudFormation "Transformation function" to automatically deploy a Lambda function responsible for rotating our database password periodically.

RDS Proxy
`Filename: /1-db-tier/db_rds_proxy.yml`

Unlike with traditional database connections, in theory, hundreds of active Lambda functions can all need an open connection to the database. To enable Lambda to reach high levels of concurrency without exhausting database connections, an RDS Proxy is deployed in front of the database cluster. For security, TLS is enabled, as well as IAM authentication.

To start, we create a role for RDS Proxy allowing it access to our Secrets Manager. Next, we create security groups for RDS Proxy and Lambda. This way we can securely give our

Lambda API function access to the proxy without having to give it direct access to the cluster.

Next we provision the RDS Proxy by bringing all of the above together. We reference the role, secret, subnets, and security group. Finally, we tell it that our Aurora cluster is the target.

Note: You can adjust the connection pool configuration to make it suitable for your business case; I used the defaults.



As you can see in our detailed design (fig.2) we connect to Secrets Manager via a VPC Endpoint. This assures that traffic does not have to traverse the internet and thus provides an extra layer of security. We will use a similar technique when working with other public services such as API Gateway and DynamoDB.

DynamoDB

`Filename: /1-db-tier/db_dynamodb_table.yml`

The configuration of our DynamoDB service is pretty straightforward. A session table called “SessionTable” is deployed with just one key called “SessionId”. Because I expect infrequent access for the sessions in my app, the STANDARD_INFREQUENT_ACCESS table class is used including PAY_PER_REQUEST billing.

Because DynamoDB is a public service, we set up a VPC endpoint with a route from the private subnets. Finally, we'll add a policy to only allow App Runner to access the service and this table only. You will notice that we'll wait until we reach the creation of the App Runner resource to create the VPC endpoint and policy. This is because we need to refer to the instance role ARN, which is not available yet at this point.

Conclusion

Now that we've completed the database tier, we can set up the layer that will connect our main app to the database: the application tier.

2.5 The Application tier

The application tier is the layer that connects our frontend to our backend. Below is a simplified version of what we are building. App Runner (deployed later as part of the web tier) will need to request an access token from Cognito to access API Gateway. API Gateway will check the provided token with Cognito and if valid will proxy the request to our Lambda API function.

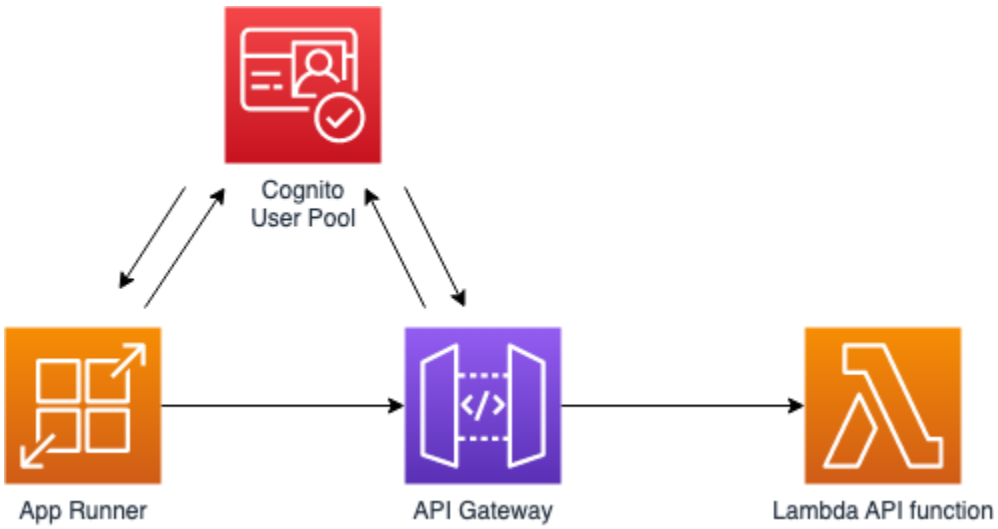


Fig 5. Logical structure of the application tier.

Cognito User Pool and User Pool Client

`Filename: /2-app-tier/cognito_userpool.yml`

Although we will be deploying an API gateway with a private endpoint, if we want to follow Zero Trust best practices, we also will want to add authentication and identity management. That's why we are adding a Cognito User Pool with associated User Pool Client to secure our API Gateway. This further reduces the risk of other resources accessing the API.

The User Pool is where we can store a username and password for App Runner to request a token for API Gateway. The User Pool Client ID tells Cognito that our App Runner is allowed to make requests for a token.

After you have deployed this stack, you will need to create a username and password combination via the console or via the AWS CLI using the following commands:

```
aws cognito-idp admin-create-user --user-pool-id <USERPOOL ID>
--username <USERNAME> --user-attributes Name=email,Value=<YOUR
EMAIL> --message-action SUPPRESS --region <YOUR REGION>

 aws cognito-idp admin-set-user-password --user-pool-id <USERPOOL
ID> --username <USERNAME> --password <PASSWORD> --permanent
--region <YOUR REGION>
```

The User pool ID can be found in the list of exports in CloudFormation and in the Web Secrets that are created later.



For more information on how to build user authentication into your application using Cognito, refer to the AWS documentation here.

Lambda API function

`Filename: /2-app-tier/app_lambda_api.yml`

Requests from API Gateway must be processed before the database can be queried, and this is where we use Lambda, a serverless compute service. To understand what's being deployed with this stack I need to explain my API development process:

Containerization

I wrote a REST API in NodeJS that runs on my local machine using the Serverless Framework. My Shopify app, a separate NodeJS application, also runs on my local machine and queries my REST API to make database calls. This setup allows me to test and develop my API locally as if it's running in AWS. Now, Using a Docker file in combination with GitHub actions, my local API is automatically "containerized" and pushed to Amazon ECR with every deployment.

Serverless Application model

Now let's look at the stack... To begin, instead of using regular CloudFormation markup, this YAML file uses AWS Serverless Application Model (SAM). SAM is a layer on top of CloudFormation that allows you to use less code to launch complex serverless applications. I'm not a big fan of SAM because I like to be in full control of what gets deployed. However, in this case it is a necessity because it's currently not possible to

connect a Lambda function to RDS Proxy with regular CloudFormation code. And this is crucial to our setup.

The Lambda function

Having said this, the provisioning of our Lambda function is now simple. We give it a name. We specify our subnets and security groups. We tell it where to find the Docker container for our NodeJS API application. We pass it a couple of environment variables so the application will know how to authenticate with RDS Proxy.

Permissions

The SAM definition will deploy 3 AWS managed policies to give Lambda permissions to connect to our VPC and execute the function. Those are: AWSLambdaVPCAccessExecutionRole, AWSLambdaExecute, and AWSLambdaBasicExecutionRole. Lastly we refer to two custom managed policies: One connects our Lambda function to the RDS Proxy. The other one allows it to write logs.



For an example on how to make a simple REST API with Node.js running on AWS Lambda and API Gateway using Serverless Framework, have a look at the following template provided by Serverless.com. If you already have a working API but it's not serverless yet, there are several publications available to help you migrate it to the Serverless Framework, like this one.

API Gateway

`Filename: /2-app-tier/app_api_gateway.yml`

API Gateway consists of many moving parts, making our YAML file longer than ideal. But each step is necessary so let's have a look at them one by one:

- First we'll create a VPC endpoint which we'll need to make sure our API Gateway can only be accessed via a private endpoint.
- Next we'll create a REST API resource and refer to the endpoint just created. It includes a policy saying that we only allow traffic from this endpoint.
- We deploy a Cognito Authorizer and attach it to the REST API.
- We tell API Gateway that it needs to proxy all requests. Meaning send all requests straight to our Lambda function.
- We deploy a method specifying that authorization via Cognito is required, and that traffic needs to be sent to a Lambda function.
- We create a role that allows API Gateway to publish logs to Cloudwatch. This will come handy when we'll need to debug for errors.

- We attach the role to the API Gateway.
- We create a log group for our access logs.
- We create a stage, attach the log group and define what we want to log. An API stage is a logical reference to a lifecycle state of your API (for example staging or production).
- Finally we tell our Lambda function that API Gateway has permission to invoke it.

Once everything has been deployed we export a https service endpoint that we can later add to our Web Secrets so App Runner knows where to send requests to.

Conclusion
Now that we've completed the application tier, we can set up the third and final layer: the Web tier.

Need help implementing this infrastructure?

E-mail me at max@skripted.io or
give me a call +1 (646) 543-2059.

2.6 The Web tier

As you've probably noticed when studying the full architecture diagram earlier, our Web tier is non conventional. When you create an App Runner service, AWS deploys your application container in an isolated, Amazon managed VPC using a Multi-AZ deployment and a load balancer. So with just a few lines of code, you'll get a highly available, scalable infrastructure.



If you're interested in learning more about how App Runner works under the hood, be sure to check out this article.

Now because App Runner is handling all the web traffic, our own "web tier" is relatively empty with only a NAT in each subnet. The reason we need NAT gateways is because we are connecting App Runner to our VPC using a "VPC Connector". The VPC connector will allow App Runner to access our API Gateway Endpoint. But this will also lead to it losing internet egress access. This can be a problem when your application needs access to the internet. The NATs will restore that.

With that explanation behind us, let's have a look at the diagram of what we are building:

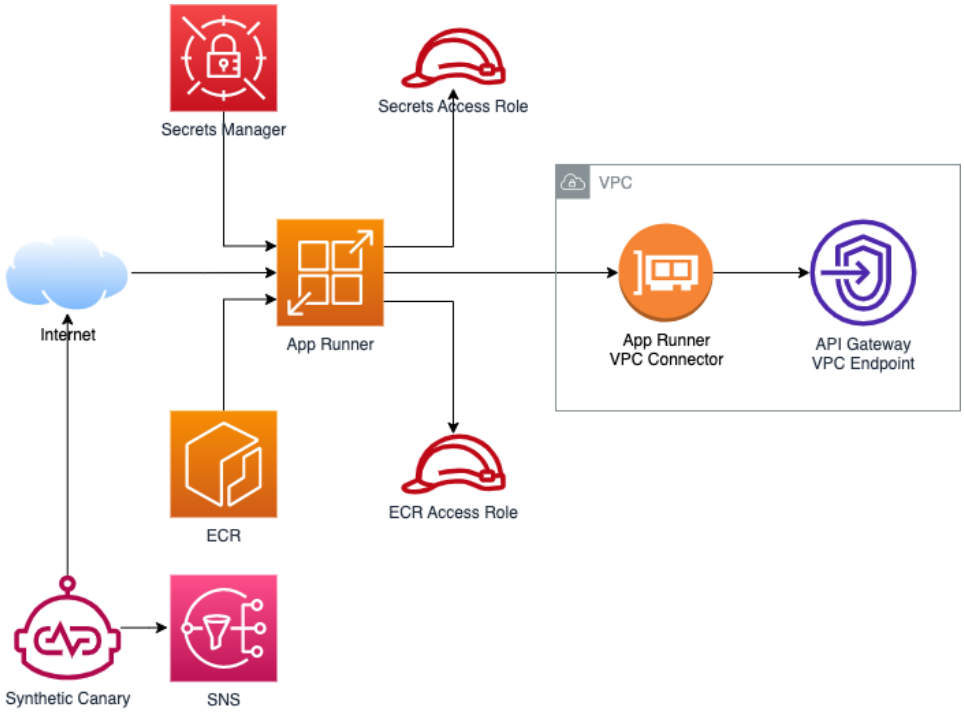


Fig 6. Logical structure of the web tier.

Deployment process

In my case, just as with my REST API, my Shopify application is containerized via Docker. So each time I make a change to the code and commit to GitHub, a GitHub action builds the image and pushes it to ECR. ECR then triggers App Runner to redeploy the application automatically. A full CI/CD process.

App Runner IAM roles

For this reason App Runner needs to assume a role that gives it access to ECR. This is done via a so-called "Access Role". My application (instance) will also need access to secrets like API and SDK keys. That's why we'll need to set up a second Secrets Manager instance and give App Runner access to it. This is done via a so-called "Instance Role".

Monitoring

Finally, to get notified when our service is down, we'll set up a Synthetic Canary via CloudWatch that will periodically poll our web endpoint and send us an email in case of an outage.

Let's start building!

Web Secrets

`Filename: /3-web-tier/web_secrets.yml`

We start by deploying our Web secrets, a second instance of a Secrets Manager resource, just like the one we used with our Database tier. It contains all the environment variables that my application needs to be able to run. For example: Node and AWS environment variables, the url to our API Gateway endpoint, the URL to our DynamoDB endpoint, Secrets for our Shopify SDK and Cognito variables like the User Pool ID, Client ID, Username and Password.

Non-sensitive variables are pulled as CloudFormation exports and automatically entered into the Secret. Others, like our Shopify SDK keys and Cognito credentials we don't want in our CloudFormation code base so you will need to enter them manually once the stack has been fully provisioned.

App Runner

`Filename: /3-web-tier/app_runner.yml`

We start our CloudFormation script by defining two roles for App Runner to assume. As explained earlier, the first is an ECR *access role* that allows App Runner to pull our container image from ECR. It references a predefined "service role" policy that includes all the policies that App Runner needs to pull the image.

Next, we define an *Instance role* describing the resources that our containerized application will need access to. In our case DynamoDB and our Web secrets. And since we now have an Instance role, we can create our VPC endpoint for DynamoDB and tell it that only App Runner is allowed to access it.

Because we need to give App Runner access to private resources like API Gateway and DynamoDB we need to configure a VPC Connector specifying our private app subnets and security group.

With all this in place we are ready to deploy our most important resource: App Runner itself. The App Runner service uses everything we described earlier. We attach the roles, describe where it can find our ECR image and secrets, and it connects to the VPC connector. Finally it describes the number of vCPUs and memory size the containers can use.

Synthetic Canary

`Filename: /3-web-tier/web_app_runner_canaryy.yml`

This infrastructure and whitepaper didn't feel complete without building some kind of observability into it. That's why the final resource we'll deploy is an automatic health check using a synthetic canary to monitor our endpoint.

First the stack creates an S3 bucket to store the artifacts (results) of every check. The script that will be executed by the canary will actually take a screenshot of the page and will output some logging information. This will all be stored in the S3 bucket for your reference.

Next a log group is created for the canary itself, so you can check if anything went wrong with the execution. Next an execution role is created that allows Synthetic Canary to deploy the script on Lambda and write to the S3 bucket and Log group. Finally, a CloudWatch alarm is deployed with an SNS topic that sends a message to your email whenever the Canary fails.

The polling period is set to every 15 minutes and Cloudwatch goes into an alarm state after two consecutive failures (so 30 minutes). You can increase or decrease these numbers by changing the parameters at the beginning of the stack.

2.7 Review

This concludes the deep dive of our 100% serverless, 3-tier infrastructure. If you run the entire CloudFormation script, it will take about 30 minutes to complete. Aurora and AppRunner take the longest to run with about 10 minutes each.

Please note that on initial creation the script will fail at the provisioning of the Lambda function. This is because you will first need to push the container image for your Lambda function to ECR. Once you've done that, simply run the script again and it will continue where it left off.

The same goes for App Runner with the additional requirement of adding the necessary API and SDK keys into your Secrets Manager manually after it has been provisioned.

That's it! I hope this whitepaper helped you get started building your own ideal stack while integrating cloud provisioning into your development workflow.

Please share your story and let me know if you have questions!

Need help implementing this infrastructure?

E-mail me at max@skripted.io or
give me a call +1 (646) 543-2059.

Appendix: Using the pre-built CloudFormation scripts

This whitepaper comes with a deployable version of the architecture in the form of a series of CloudFormation scripts (called stacks). The stacks are designed to be deployed in a multi-account environment. For example, you can use the same collection of stacks to deploy a staging environment in one account and then an exact copy in a production account.

For this reason, some files have variables at the start of the document. I.e. it is advised to have a different VPC CIDR block for each account so you can apply VPC peering later if desired. You also might want to include different environment variables per account. Etc.

CI/CD using GitHub Workflows

Because good infrastructure management is a continuous process like software development, the stacks have been prepared for use in a CI/CD pipeline using a GitHub workflow. This allows you to make incremental changes to the CloudFormation code and automatically update your infrastructure with every push to GitHub.

To make your life easier, a GitHub Workflow document is provided in the ./github folder. You can make copies of the document for each environment needed and then update the variables at the top of the document as necessary. Then, once you push changes to let's say your staging branch, those changes will automatically be deployed to your staging environment.

Pre-requisites

Setting up federated AWS authentication
The actions in the workflow use federated AWS authentication instead of access keys and secrets. This is safer than storing secrets in GitHub. And because federated access keys are only valid for a few hours you won't have to worry about key rotation.

You can read more about this process here:
https://docs.github.com/en/actions/deployment/security-hardening-your-deployments/configuring-openid-connect-in-amazon-web-services

Creating the GitHub Role in IAM
The way federated AWS authentication works is that you need to create an IAM role that can be assumed by GitHub to deploy your CloudFormation script onto your AWS account. This role needs explicit permissions to provision the resources the stack depends on. Also, it should follow the principle of least privileges.

To help you get started, a CloudFormation script is provided that creates this role for you. It's called `iam_role_for_github.yml`. If set up correctly, this is the only script you actually need to manually deploy using an IAM user that has the `AdministratorAccess` managed policy attached. Next you should be able to deploy the rest of the script using the provided GitHub actions.



***Caution**: once the role is created, in theory anyone with access to the full role ARN will be able to assume it and make changes to your account. To prevent this from happening, the iam_role_for_github.yml script includes setting up a "trust relationship" with your GitHub repository. This way the role can only be assumed from your repository. To make sure it works correctly, update the GitHubOrg parameter to your repository where "yourorg/*" means "all repositories in the "yourorg" GitHub account.*

Step by step

The entire process can be broken down in the following steps:

```
_
# 1. Create a fresh AWS account.

# 2. In IAM using the console, set up AWS keyless federation:

1. Go to IAM console
2. Click "Identiy providers"
3. Click "Add provider"
4. Select "OpendID Connect" with provider URL:
https://token.actions.githubusercontent.com
5. Click "Get thumbprint"
6. For "Audience" type: "sts.amazonaws.com" and click "Add provider"
7. Done.
```

 # 3. Create the IAM role for GitHub

1. In IAM, in the “Identity Providers” overview click "token.actions.githubusercontent.com"
2. Copy the generated arn, ie: "arn:aws:iam::01234567890:oidc-provider/token.actions.githubusercontent.com"
3. Paste arn into the "iam_role_for_github.yml" script of this repository.
4. Update the parameter called “GitHubOrg” to your repository
5. Manually deploy iam_role_for_github.yml in CloudFormation

For more information on setting up keyless github access visit: https://docs.cloudquery.io/blog/keyless-access-to-aws-in-github-actions-with-oidc

For more information on setting up OpenID in AWS visit https://docs.github.com/en/actions/deployment/security-hardening-your-deployments/configuring-openid-connect-in-amazon-web-services

 # 4. Update GitHub workflow script.

1. The previous script outputs the role arn. Copy this arn.
2. Paste this arn into ./github/workflows/deploy_to_aws.yml

 # 5. Push to staging or master

Assuming you are working in the “staging” branch, and your GitHub workflow is active in ./github/workflows, once you push your branch the workflow will be automatically deployed. Repeat step #4 and #5 for each branch and AWS account you need to set up.

Copyright 2023 by Maxim Schram / Skripted. All Rights Reserved.

THIS WHITEPAPER IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE WHITEPAPER OR THE USE OR OTHER DEALINGS IN THE WHITEPAPER.