serverlessland.com

BUILDING SERVERLESS APPLICATIONS WITH TERRAFORM

Anton Aleksandrov and Debasis Rath

Building serverless applications with Terraform

Anton Aleksandrov and Debasis Rath

v0.0.4

Table of Contents

1. Introduction

Infrastructure-as-code (IaC) is a methodology of applying techniques commonly used for source code to infrastructure templates. IaC allows engineering teams to define required infrastructure in either a declarative or imperative way, and enables capabilities such as version control, audit, and CI/CD.

Hashicorp Terraform is an IaC tool that allows engineering teams to define and deploy their applications using a declarative, human-readable syntax. This guide is targeting engineering teams that are looking for guidance and samples for using Terraform to build serverless applications.

While this guide covers some of the Terraform basics, it assumes you do have previous experience and some level of proficiency with Terraform.

Overview

One of the major benefits of using IaC is the fact that it allows to easily create multiple environments from the same infrastructure template. This helps engineering teams to ensure that different application environments can be easily replicated and kept in sync, reducing the risk of application functionality breaking. This is applicable to multiple scenarios. One common scenario is using IaC to deploy development, testing, and production environments. Another example is using IaC to deploy a set of dedicated data plane assets in a SaaS application, where each tenant has their own set of single-tenant infrastructure resources. IaC reduces the risk of human error and allows to automate replicating changes across environments.

Over the years IaC practices and tools evolved to cover thousands of various resource types, many of them going beyond the original definition of "infrastructure". For example, today you can use IaC to manage resources such as user accounts or even application settings. Serverless applications are no different. Building Serverless applications with IaC tools is a common task, and we highly recommend you use IaC tools for building your Serverless applications on AWS.

Infrastructure-as-code tools

There are various IaC tools engineering teams use for building Serverless applications. AWS provides IaC tools such as AWS Serverless Application Model (SAM), and AWS Cloud Development Kit (CDK). Other IaC tools, such as Hashicorp Terraform, are provided by AWS Partners or 3rd party vendors. Selecting which tool to use depends on several factors, such as whether you want to use declarative or imperative approach, or whether your organization is already using a particular IaC methodology and you'd like to reuse the knowledge.

AWS SAM is an open-source framework for building serverless applications. SAM is a superset of AWS CloudFormation, and provides a shorthand syntax to express multiple types of serverless resources, such as functions, APIs, databases, and event source mappings. With SAM, you model your environments using YAML, which is automatically converted to CloudFormation syntax and deployed to AWS using the SAM CLI. SAM also provides local tools, allowing engineers to test their functions locally or simulate an API Gateway.

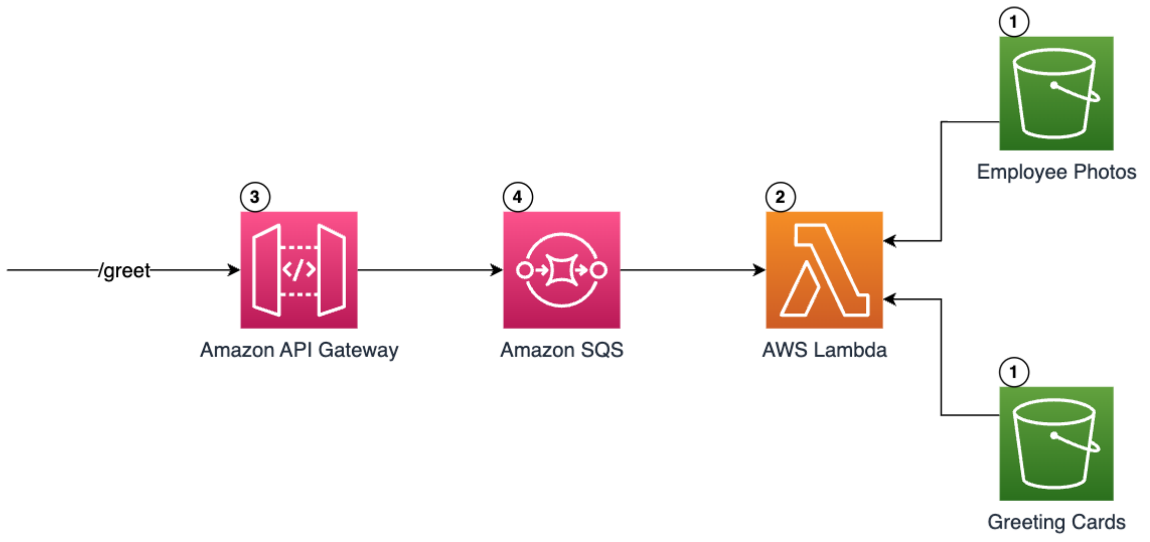
AWS CDK takes a programmatic approach. With CDK you model your environments using one of the supported programming languages, such as TypeScript, Python, or Java. Similar to SAM, the CDK CLI will automatically convert your environment definition to CloudFormation syntax and deploy it to AWS. CDK can be used together with SAM, allowing engineers to use SAM for local testing when their environments are defined with CDK.

Hashicorp Terraform is a popular IaC tool coming from an AWS Partner, widely adopted in the industry. With Terraform you use the HashiCorp Terraform Language (HCL) to write Terraform configuration files (.tf). You use those files to model your environments. Similar to CDK, Terraform can also integrate with SAM when building Serverless applications for ease of local testing, as you will see in this guide.

Throughout this guide you will learn how to use Terraform for building serverless applications. You will follow a process of gradually building and evolving serverless application architecture. You will learn how to address IAM and observability, and how to use SAM together with Terraform for testing your applications locally. Towards the end of this guide you will learn how to scale-up your Terraform templates for serverless applications by adding support for multiple environments and modularizing infrastructure and application code.

2. Solution Overview

Throughout this guide, you will learn how to use Terraform to build a Serverless application that is used to generate greeting cards for employees working in a fictitious enterprise named AcmeCorp. In the final chapter you can download the sample code used in this guide, as well as get a collection of actionable resources to continue your learning journey.



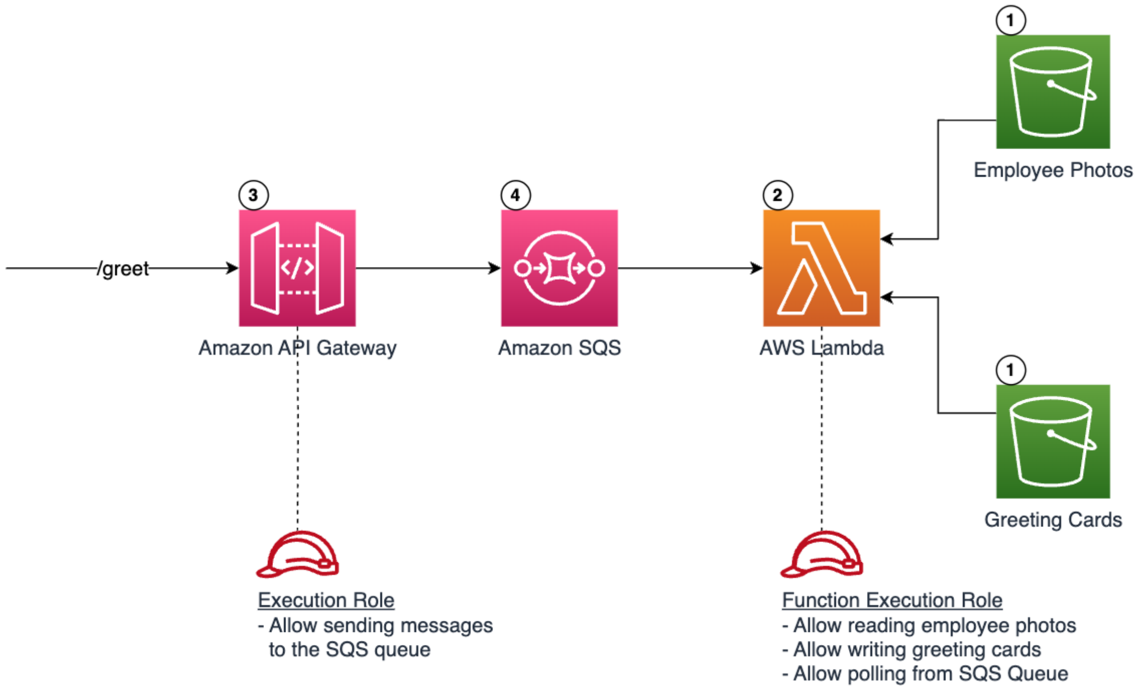
Each chapter of this guide builds on the previously created solution, adds a new concept, and explains how to wire it with existing components. Note that this guide is not intended to be consumed as a step-by-step workshop or tutorial, its purpose is to illustrate and provide guidance through the concepts you can use to create your own Serverless applications.

- **Chapter 3** illustrates configuring Terraform and using it to create Amazon S3 buckets - one for storing employee photos (source bucket), the other for storing generated greeting cards (target bucket).
- **Chapter 4** shows how to create an AWS Lambda function that receives an array of employee IDs and generates greeting cards. The function will retrieve photos from the source bucket, generate greeting cards, and store them in the target bucket. Initially you'll invoke that function manually using the AWS CLI.
- **Chapter 5** explains how to make the application more consumable by other parts of the business by exposing the functionality via Amazon API Gateway.
- **Chapter 6** introduces additional resiliency and scalability into the architecture by adding an Amazon SQS queue to generate greeting cards asynchronously. This would help during the holiday season, when the app needs to generate greeting cards for thousands of employees.
- **Chapter 7 and 8** focus on operational aspects of the application, such as observability, deploying to multiple environments and more.

Identity and Access Management

When building applications on AWS, it is important you take care of IAM permissions in order to only allow resources you permit to communicate with each other. For example, you will need to grant your Lambda function permissions to poll (but not push) messages from the SQS queue, as well as read from and write to S3 buckets.

As a general practice, the guide follows the principle of least privileged access. Each step demonstrates how to grant the minimal required set of permissions. The guide will demonstrate how to use a combination of execution roles and resource policies. For example, the Lambda function will have an execution role allowing it to poll messages from the SQS Queue, and SQS Queue will have a resource policy allowing API Gateway to send messages. Read more about using AWS IAM.


1
Employee Photos
3
4
2
/greet
Amazon API Gateway
Amazon SQS
AWS Lambda
1
Greeting Cards
Execution Role
- Allow sending messages to the SQS queue
Function Execution Role
- Allow reading employee photos
- Allow writing greeting cards
- Allow polling from SQS Queue

Tools

Below tools are used throughout the guide. Install them on your machine if you want to replicate steps described in this guide.

- AWS CLI - installed and configured to work with an AWS Region of your choice
- Terraform CLI
- Text editor or IDE of your choice, such as Visual Studio Code.

Let's get started!

3. S3 buckets

Chapters 3-6 of this guide illustrate how to configure the AWS Terraform Provider, create resources, and wire application components together. For simplicity, in these chapters the guide assumes you're using a single Terraform configuration file named `main.tf`. Towards the end of this guide you will learn how to organize your project into an easy-to-maintain, modular multi-environment solution that would work for a large enterprise.

Adding the AWS Terraform Provider

AWS Terraform Provider allows you to manage the lifecycle of many resource types supported by AWS. The following snippet adds the AWS Terraform Provider to your Terraform project.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

provider "aws" {
  region = "us-east-1" # Replace with your desired region
}
```

AWS Terraform provider requires valid AWS credentials in order to create and manage resources in your AWS account. There are several ways to achieve this.

Environment Variables You can export AWS access and secret keys as environment variables. This is the recommended method, as it keeps secrets out of your codebase.

```
export AWS_ACCESS_KEY_ID="your-access-key"
export AWS_SECRET_ACCESS_KEY="your-secret-access-key"
```

AWS Terraform provider will automatically recognize these environment variables in runtime.

Explicit Configuration You can also explicitly provide these credentials in the Terraform configuration file.

> WARNING: Hardcoding credentials is not recommended. It exposes you to secrets leakage in case you commit your files to a public source control system.

```
provider "aws" {
  region  = "us-east-1"
  access_key = "your-access-key"
  secret_key = "your-secret-access-key"
}
```

Creating S3 Buckets

The syntax you use for creating resources with Terraform is

```
resource <resource_type> <resource_name> {
  ... resource properties ...
}
```

Below snippet creates an S3 bucket. This bucket is used for storing the source images. Ensure the bucket name is globally unique to avoid any naming conflicts. In addition to the bucket, the snippet also creates two additional resources to ensure that the bucket is private, and cannot be accessed without proper permissions.

```
resource "aws_s3_bucket" "src_bucket" {
  bucket = "src-bucket-some-random-string"
}

resource "aws_s3_bucket_ownership_controls" "src_bucket_ownership_controls" {
  bucket = aws_s3_bucket.src_bucket.id
  rule {
    object_ownership = "BucketOwnerPreferred"
  }
}

resource "aws_s3_bucket_acl" "src_bucket_acl" {
  depends_on = [aws_s3_bucket_ownership_controls.src_bucket_ownership_controls]
  bucket = aws_s3_bucket.src_bucket.id
  acl    = "private"
}
```

Duplicate the above Terraform configuration snippet to create another S3 bucket named `dst_bucket`. This bucket will be used to store the generated greeting cards.

The following commands will initialize a new Terraform project, load all the required dependencies, such as AWS Terraform Provider declared earlier, and deploy resources to your AWS account.

```
# Initiates the project, loads required modules
terraform init

# Creates the change set, and gives you an overview of what you are deploying
terraform plan

# Deploys the actual resources on to your AWS Account
terraform apply
```

Once successfully deployed, you will see an output similar to below:

```
aws_s3_bucket.dst_bucket: Creating...
aws_s3_bucket.src_bucket: Creating...
aws_s3_bucket.dst_bucket: Creation complete after 16s
```

```
aws_s3_bucket.src_bucket: Creation complete after 18s
.....
```

Uploading photos to the source S3 bucket

Once the `src_bucket` is created, use the Amazon S3 Console to upload an image. You can also do it with `aws s3 cp` command using the AWS CLI.

The remainder of this guide assumes you've uploaded an image to the source S3 bucket named `johnsmith.jpg`, where `johnsmith` is the employee ID.

Recap

In this chapter you've learned how to:

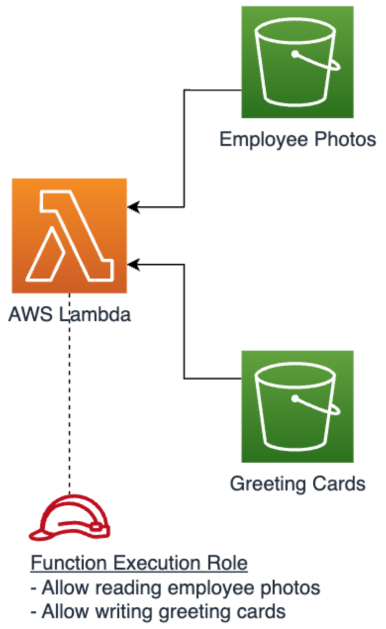
1. Add the Terraform provider to your `main.tf`
2. Configure different methods of providing AWS credentials to Terraform, with emphasis on keeping the secrets secure.
3. Create S3 Buckets configured for private access only.
4. Initialize the Terraform and deploy your Terraform configuration to your AWS account.

Next step

In the next chapter you will add a Lambda function that generates greeting cards. You will grant it required IAM permissions in order to retrieve images from the source bucket and upload generated greeting cards to the destination bucket.

4. Lambda functions

In the previous chapter you've learned how to configure and use Terraform to create the source and destination S3 buckets. In this chapter you will learn to create AWS Lambda functions. The function illustrated in this chapter processes photos from the source bucket, generates greeting cards, and uploads them to the destination bucket.



Creating IAM role and access policy

Lambda function requires an IAM execution role with relevant access permissions. Below snippet creates a role and assigns it a policy with permissions to read from the source bucket and write to the destination bucket.

```
# Execution role
resource "aws_iam_role" "lambda_execution_role" {
  name = "terraform-lambda-greetings-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17",
    Statement = [
      {
        Action = "sts:AssumeRole",
        Principal = {
          Service = "lambda.amazonaws.com"
        },
        Effect = "Allow",
        Sid    = ""
      }
    ]
  })
}

# Access policy
resource "aws_iam_policy" "lambda_s3_access_policy" {
  name        = "terraform-lambda-s3-access-policy"
  description = "Grants access to source and destination buckets"

  policy = jsonencode({
    Version = "2012-10-17",
    Statement = [
      {
        Action   = ["s3:GetObject"],
        Effect   = "Allow",
        Resource = [
          "${aws_s3_bucket.src_bucket.arn}/*"
        ]
      },{
        Action   = ["s3:PutObject"],
        Effect   = "Allow",
        Resource = [
          "${aws_s3_bucket.dst_bucket.arn}/*"
        ]
      }
    ]
  })
}

# Attaches the policy to the role
resource "aws_iam_role_policy_attachment" "s3_full_access_attachment" {
  policy_arn = aws_iam_policy.lambda_s3_access_policy.arn
  role       = aws_iam_role.lambda_execution_role.name
}
```

Creating a Lambda Function

Find the `index.mjs` file under Extra Resources on this page. It contains the sample source code of a Lambda function that retrieves an image from the source bucket, creates an HTML greeting card, and uploads it to the destination bucket.

Terraform provides the `archive_file` data type which can be used to package `index.mjs` into a ZIP file. This ensures that Lambda functions consistently receive the latest code version, reducing manual steps and promoting reliable, automated updates.

The code snippet below builds a ZIP archive, and uses it for creating a Lambda function. Note that two environment variables that contain references to source and destination bucket names are defined for the function. Those environment variables are used in the function code. The `source_code_hash` property ensures that the Lambda function resource is updated with the latest code version whenever any code changes are detected.

```
# Create a zip file with function code
data "archive_file" "lambda_zip" {
  type        = "zip"
  source_file = "index.mjs"
  output_path = "lambda.zip"
}

# Create a Lambda function
resource "aws_lambda_function" "greeting_lambda" {
  function_name = "greetings-lambda-function"

  handler     = "index.handler"
  runtime     = "nodejs18.x"
  memory_size = 256
  role        = aws_iam_role.lambda_execution_role.arn

  environment {
    variables = {
      SRC_BUCKET = aws_s3_bucket.src_bucket.id,
      DST_BUCKET = aws_s3_bucket.dst_bucket.id
    }
  }

  filename         = data.archive_file.lambda_zip.output_path
  source_code_hash = data.archive_file.lambda_zip.output_base64sha256
}
```

Run `terraform apply` to deploy the updated application. Running `terraform plan` before `terraform apply` is a recommended best practice to preview the changes Terraform will make based on your updated configurations.

> The updated configuration uses the new `archive_file` data type, you might be asked to run `terraform init --upgrade` to install the required dependencies.

Testing the Lambda function locally using SAM

A common practice during development is to test the code functionality locally. Developers commonly test their code locally before transitioning to a live cloud environment. While Terraform excels in infrastructure setup, it does not natively provide support for testing Lambda functions locally. AWS SAM is a unique tool to test your Lambda function locally, ensuring everything runs smoothly before deploying to the cloud. This blog talks in detail about testing your Lambda function locally when using Terraform for IaC.

Ensure that your SAM CLI is up-to-date with the latest version. Create an `event.json` file with below JSON

```
{
  "body": "{\"employeeId\": \"johnsmith\"}"
}
```

Run the below command:

```
sam local invoke \
  --hook-name terraform \
  greeting_lambda \
  -e event.json
```

Output:

```
{
  "statusCode": 200,
  "body": "{\"message\":\"Processed successfully\"}"
}
```

Invoking the Lambda function running on AWS using the AWS CLI

The following snippet demonstrates how to use AWS CLI to invoke the Lambda function. It uses the same `event.json` file created in the previous section.

```
aws lambda invoke \
  --function-name greeting_lambda \
  --cli-binary-format raw-in-base64-out  \
  --payload file://event.json \
  output.txt
```

This will invoke the Lambda function and store the result in `output.txt`.

You will see a newly generated greeting card named `greeting-card-johnsmith.html` in the destination S3 bucket.

You can use SAM to browse the CloudWatch logs generated during function invocation

```
sam logs --cw-log-group /aws/lambda/greeting_lambda
```

Recap

In this chapter you've learned how to:

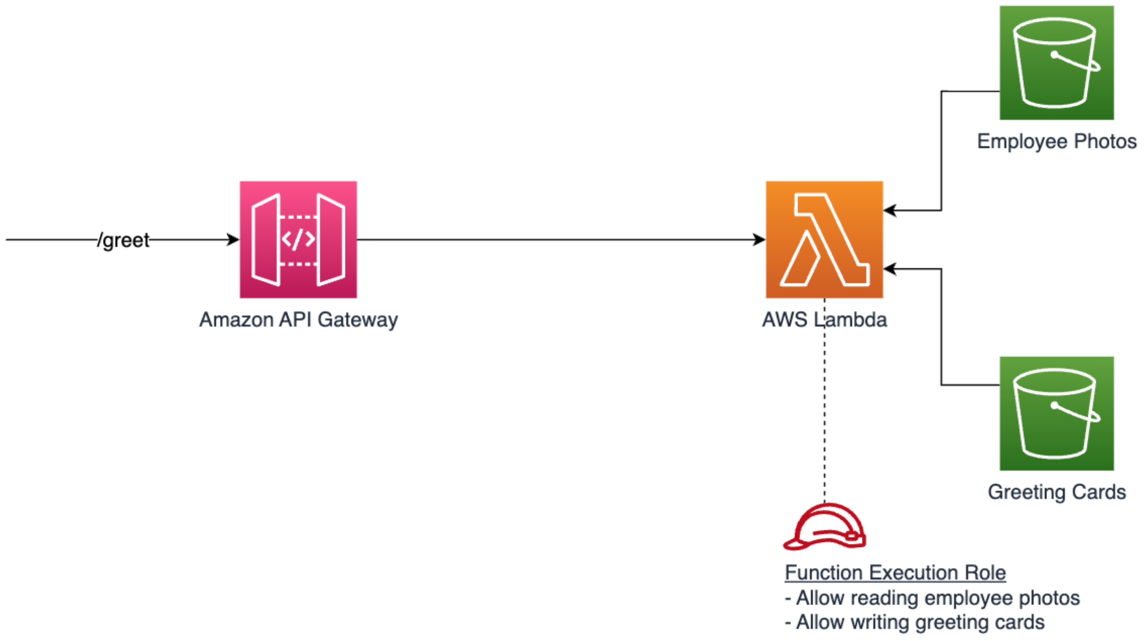
1. Create a Lambda function execution role and assign access policies
2. Create a Lambda function
3. Test your function locally
4. Invoke Lambda function in the cloud using the AWS CLI.

Next step

In the next chapter you will learn how to set up an API Gateway to expose your Lambda function, allowing external clients to invoke your function via a public API.

5. API Gateway

The ability to invoke a Lambda function through an HTTPS request opens up many integration opportunities. This chapter will illustrate creating a public RESTful API for a Lambda function using the Amazon API Gateway, as well as testing the API locally.



Setting up the Amazon API Gateway

The following snippet creates a regional REST API Gateway, defines a RESTful resource, sets access permissions, and integrates the resource with Lambda function.

```
# Create a REST API Gateway
resource "aws_api_gateway_rest_api" "greeting_api" {
  name        = "greeting_api"
  description = "API for invoking the Greeting Lambda Function"
  endpoint_configuration {
    types = ["REGIONAL"]
  }
}

# Create an API Resource
resource "aws_api_gateway_resource" "greet_resource" {
  rest_api_id = aws_api_gateway_rest_api.greeting_api.id
  parent_id   = aws_api_gateway_rest_api.greeting_api.root_resource_id
  path_part   = "greet"
}
```

```
# Create an API Method
resource "aws_api_gateway_method" "greet_method" {
  rest_api_id   = aws_api_gateway_rest_api.greeting_api.id
  resource_id   = aws_api_gateway_resource.greet_resource.id
  http_method   = "POST"
  authorization = "NONE"
}

# Grant API Gateway permissions to invoke the Lambda function
resource "aws_lambda_permission" "allow_api_gateway" {
  statement_id  = "AllowAPIGatewayInvoke"
  action        = "lambda:InvokeFunction"
  function_name = aws_lambda_function.greeting_lambda.function_name
  principal     = "apigateway.amazonaws.com"

  # This will only allow a specific API resource and method to invoke the Lambda function.
  # Otherwise any API Gateway in the account will be able to invoke the Lambda function.
  # This is required for adhering to least privileged access principle
  source_arn = "${aws_api_gateway_rest_api.greeting_api.execution_arn}/*/${aws_api_gateway_method.g
}

# Integrate API Gateway with the Lambda function
resource "aws_api_gateway_integration" "greet_lambda_integration" {
  rest_api_id = aws_api_gateway_rest_api.greeting_api.id
  resource_id = aws_api_gateway_resource.greet_resource.id
  http_method = aws_api_gateway_method.greet_method.http_method

  integration_http_method = "POST"
  type                    = "AWS_PROXY"
  uri                     = aws_lambda_function.greeting_lambda.invoke_arn

  depends_on = [aws_lambda_permission.allow_api_gateway]
}

# Create a new API Gateway deployment
resource "aws_api_gateway_deployment" "greeting_api_deployment" {
  rest_api_id = aws_api_gateway_rest_api.greeting_api.id
  stage_name  = "prod"

  triggers = {
    redeployment = sha256(jsonencode(aws_api_gateway_rest_api.greeting_api.body))
  }

  lifecycle {
    create_before_destroy = true
  }
  depends_on = [aws_api_gateway_method.greet_method]
}

# Output the API Gateway invocation endpoint
output "greeting_api_endpoint" {
  value = "${aws_api_gateway_deployment.greeting_api_deployment.invoke_url}/greet"
}
```

Running `terraform apply` will deploy the updated application.

Testing the API locally with SAM

Run the following command from the directory where the `main.tf` file is located.

```
sam local start-api --hook-name terraform
```

This is what output will look like

```
Mounting greeting_lambda at http://127.0.0.1:3000/greet [POST]
You can now browse to the above endpoints to invoke your functions.
You do not need to restart/reload SAM CLI while working on your
functions, changes will be reflected instantly/automatically.
If you used sam build before running local commands, you will need
to re-run sam build for the changes to be picked up. You only need
to restart SAM CLI if you update your AWS SAM template
2023-10-06 18:35:33 WARNING: This is a development server. Do not
use it in a production deployment. Use a production WSGI server
instead.
 * Running on http://127.0.0.1:3000
```

Use curl, Postman, or a similar tool to send requests to the local API endpoint. The API will call the local Lambda function instance which is instantiated by SAM in the local environment. To know more about how SAM makes it happen visit this url.

```
curl  -v http://127.0.0.1:3000/greet \
      -H 'Content-Type:application/json' \
      -d '{"employeeId":"johnsmith"}'
```

Deploying and testing the API in cloud environment

You can use Terraform outputs to print out the newly created API Gateway invocation endpoint.

```
output "greeting_api_endpoint" {
  value = "${aws_api_gateway_deployment.greeting_api_deployment.invoke_url}/greet"
}
```

```
Outputs:

greeting_api_endpoint = "https://{api-id}.execute-api.{region}.amazonaws.com/prod/greet"
```

Use curl, Postman, or similar tools to send requests to the API Gateway endpoint.

```
curl -X POST \
     -H 'Content-Type: application/json' \
     -d '{"employeeId":"johnsmith"}' \
     https://<api-id>.execute-api.<region>.amazonaws.com/prod/greet
```

Replace `<api-id>` and `<region>` with the values you got in the previous step.

You will see a newly generated greeting card named `greeting-card-johnsmith.html` in the destination S3 bucket.

Recap

In this chapter you've learned how to:

1. Create a RESTful API Gateway
2. Define an API Gateway resource with a POST method
3. Define access permission and integrate the API Gateway with a Lambda function
4. Test the API locally
5. Deploy and test the API in cloud

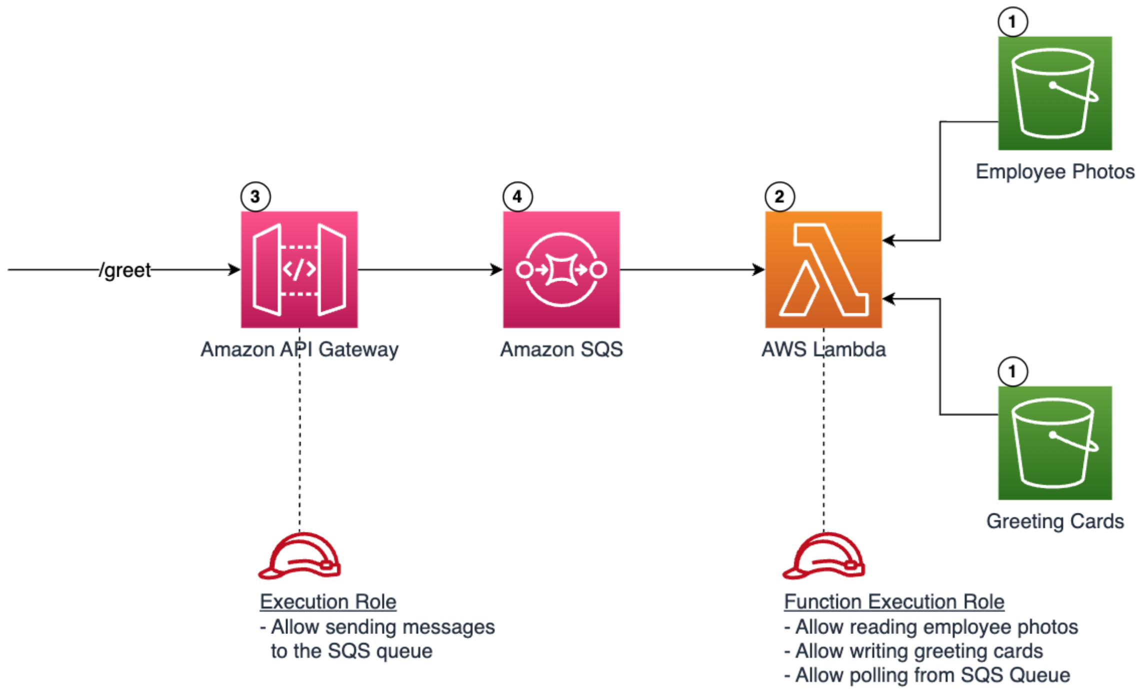
Next Step

The next chapter will add scalability and resilience to the application architecture by introducing an SQS queue between API Gateway and Lambda function. Instead of sending requests to the Lambda function directly, the API Gateway will put messages to the queue instead, and Lambda function will process messages asynchronously by polling from the queue.

6. Asynchronous processing

The previous chapter described how to directly integrate API Gateway with a Lambda function. With this approach, the Lambda function will be invoked synchronously each time API Gateway receives an incoming `/greet` request.

This chapter describes a more resilient and scalable approach, where Amazon SQS queue is used as an intermediary to decouple request intake from processing, ensuring better scalability and error-handling. This asynchronous pattern is recommended when you want to control the processing flow, and protect your downstream dependencies from overflow. For example, during the holiday season there might be tens of thousands of requests to generate greeting cards, but one of the downstream 3rd party dependencies is unable to process requests at this rate. Decoupling the workflow using an SQS queue allows the application to receive large volumes of incoming requests, buffer them in the message queue, and process at a controlled rate.



Creating an Amazon SQS queue

The following snippet creates an SQS queue. Note that data encryption is enabled for this queue for extra security.

```
resource "aws_sqs_queue" "greeting_queue" {
  name                    = "greetings_queue"
  sqs_managed_sse_enabled = true
}
```

Configuring API Gateway

The following snippet configures the API Gateway to send requests as messages to an SQS queue, instead of sending directly to a Lambda function.

```
# Create an IAM Role for API Gateway
resource "aws_iam_role" "api_gateway_greeting_queue_role" {
  name = "api_gateway_greeting_queue_role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17",
    Statement = [
      {
        Action = "sts:AssumeRole",
        Principal = {
          Service = "apigateway.amazonaws.com"
        },
        Effect = "Allow"
        Sid    = ""
      }
    ]
  })
}

# Create a policy allowing API Gateway to send messages to the SQS queue
resource "aws_iam_role_policy" "api_gateway_greeting_queue_role_policy" {
  name = "api_gateway_greeting_queue_role_policy"
  role = aws_iam_role.api_gateway_greeting_queue_role.name

  policy = jsonencode({
    Version = "2012-10-17",
    Statement = [
      {
        Action   = "sqs:SendMessage",
        Effect   = "Allow",
        Resource = aws_sqs_queue.greeting_queue.arn
      }
    ]
  })
}

data "aws_region" "current" {}
data "aws_caller_identity" "current" {}

# Create an integration that sends incoming request body as a message to SQS
resource "aws_api_gateway_integration" "greet_method_integration" {
  rest_api_id             = aws_api_gateway_rest_api.greeting_api.id
  resource_id             = aws_api_gateway_resource.greet_resource.id
  http_method             = aws_api_gateway_method.greet_method.http_method
  type                    = "AWS"
  integration_http_method = "POST"
  uri                     = "arn:aws:apigateway:${data.aws_region.current.name}:sqs:path/${data.aws_
  request_parameters = {
    "integration.request.header.Content-Type" = "'application/x-www-form-urlencoded'"
  }
  request_templates = {
    "application/json" = "Action=SendMessage&MessageBody=$input.body"
  }
  credentials = aws_iam_role.api_gateway_greeting_queue_role.arn
}

resource "aws_api_gateway_integration_response" "integration_response_200" {
  rest_api_id = aws_api_gateway_rest_api.greeting_api.id
  resource_id = aws_api_gateway_resource.greet_resource.id
  http_method = aws_api_gateway_method.greet_method.http_method
  status_code = 200
  selection_pattern = "^2[0-9][0-9]" # Any 2xx response

  response_templates = {
    "application/json" = "{\"status\": \"success\"}"
  }

  depends_on = [aws_api_gateway_integration.greet_method_integration]
}

resource "aws_api_gateway_method_response" "method_response_200" {
  rest_api_id = aws_api_gateway_rest_api.greeting_api.id
  resource_id = aws_api_gateway_resource.greet_resource.id
  http_method = aws_api_gateway_method.greet_method.http_method
  status_code = 200

  response_models = {
    "application/json" = "Empty"
  }
}
```

Configuring Lambda event source mapping

The following snippet configures the Lambda function to poll messages from the queue for processing.

```
# Create a policy with permissions required to poll messages from an SQS queue
resource "aws_iam_policy" "greeting_lambda_sqs_policy" {
  name        = "greeting_lambda_ssqs_policy"
  description = "Grants access to read messages from SQS"
  policy = jsonencode({
    Version = "2012-10-17",
    Statement = [
      {
        Action = ["sqs:ReceiveMessage", "sqs:DeleteMEssage", "sqs:GetQueueAttributes"],
        Effect = "Allow",
        Resource = [aws_sqs_queue.greeting_queue.arn]
      }
    ]
  })
}

# Attach the policy to Lambda execution role created previously
resource "aws_iam_role_policy_attachment" "greeting_lambda_sqs_policy_attachment" {
  policy_arn = aws_iam_policy.greeting_lambda_sqs_policy.arn
  role       = aws_iam_role.greeting_lambda_execution_role.name
}

# Create a Lambda event-source mapping to enable Lambda to poll from the queue
resource "aws_lambda_event_source_mapping" "greeting_sqs_mapping" {
  event_source_arn = aws_sqs_queue.greeting_queue.arn
  function_name    = aws_lambda_function.greeting_lambda.function_name
  batch_size       = 1

  depends_on = [aws_iam_role_policy_attachment.greeting_lambda_sqs_policy_attachment ]
}
```

Deploying and testing the application in cloud

After running `terraform apply` to apply the changes to the cloud environment, you will see an output containing the API Gateway invocation endpoint

```
Outputs:

greeting_api_endpoint = "https://{api-id}.execute-api.{region}.amazonaws.com/prod/greet"
```

Use curl, Postman, or similar tools to send requests to the API Gateway endpoint.

```
curl -X POST \
     -H 'Content-Type: application/json' \
     -d '{"employeeId":"johnsmith"}' \
     https://<api-id>.execute-api.<region>.amazonaws.com/prod/greet
```

Replace `<api-id>` and `<region>` with the values you got in the previous step.

You will see a newly generated greeting card named `greeting-card-johnsmith.html` in the destination S3 bucket.

Recap

In this chapter you've learned how to:

1. Create an SQS queue and set IAM permissions
2. Create an API Gateway integration to send messages to the SQS queue
3. Create Lambda policy to allow polling messages from the SQS queue
4. Create Lambda integration to process messages from the SQS queue
5. Deploy and test the API in cloud

By introducing SQS, you've enhanced our system's scalability, as well as improved its resilience and error-handling capacity.

Next Step

The next chapter will illustrate enabling observability capabilities with Terraform.

7. Observability

In this chapter you will learn about adding observability capabilities to your serverless applications using Terraform and IaC. Observability is a vital trait of any cloud application, as it provides insights into system behavior and interactions, aiding in issue detection and performance optimization. It is common to name Logging, Traces, and Metrics as three pillars of observability.

Another crucial aspect of observability is tagging. Tags assist in organizing resources, enable precise cost tracking, and when used with a Configuration Management Database (CMDB), help manage assets and ensure compliance. In a cloud environment, where resources and configurations are dynamic, a structured observability approach, supported by effective tagging, is essential for operational efficiency and cost management.

Lambda and CloudWatch Logs

AWS Lambda natively integrates with CloudWatch Logs, pushing all logs from your code to a CloudWatch Logs group. Make sure that the execution role you assign to your functions have permissions to create log groups and write log events to CloudWatch Logs:

```
resource "aws_iam_policy" "lambda_cloudwatch_policy" {
  name        = "LambdaCloudWatchPolicy"
  description = "Allows Lambda to write logs to CloudWatch."

  policy = jsonencode({
    Version = "2012-10-17",
    Statement = [
      {
        Action   = ["logs:CreateLogGroup", "logs:CreateLogStream", "logs:PutLogEvents"],
        Effect   = "Allow",
        Resource = "arn:aws:logs:{region}:{account-id}:log-group:{log-group-name}"
      }
    ]
  })
}

resource "aws_iam_role_policy_attachment" "lambda_cloudwatch_attach" {
  policy_arn = aws_iam_policy.lambda_cloudwatch_policy.arn
  role       = aws_iam_role.lambda_execution_role.name
}
```

Once you grant the permissions, the Lambda service will automatically create and update the Log Groups and streams for you.

API Gateway and CloudWatch Logs

API Gateway supports two types of logging - execution and access. With execution logging, API Gateway will create and manage the log group for you. You will need to configure the API Gateway with an execution role that has the required permissions.

```
resource "aws_api_gateway_account" "api_gateway_account" {
  cloudwatch_role_arn = aws_iam_role.cloudwatch.arn
}
```

See aws_api_gateway_account for a full example.

In order to enable access logging for API Gateway, you need to create a CloudWatch Log Group, ensure the IAM role used by the API Gateway has permissions to publish logs, and enable the access logs for the API Gateway:

```
# Create a CloudWatch Logs Group
resource "aws_cloudwatch_log_group" "api_gateway_log_group" {
  name = "/aws/api-gateway/greeting_api"
}

# Create an access policy for API Gateway to send execution logs to CloudWatch Logs Group
resource "aws_iam_role_policy" "api_gateway_cloudwatch_logs" {
  name   = "ApiGatewayCloudWatchLogs"
  role   = aws_iam_role.api_gateway_execution_role.name

  policy = jsonencode({
    Version = "2012-10-17",
    Statement = [
      {
        Action   = ["logs:PutLogEvents", "logs:CreateLogStream"],
        Effect   = "Allow",
        Resource = aws_cloudwatch_log_group.api_gateway_log_group.arn
      }
    ]
  })
}

# Enable access logs in your API Gateway
resource "aws_api_gateway_stage" "api_gateway_stage" {
  # ... (other configurations)

  access_log_settings {
    destination_arn = aws_cloudwatch_log_group.api_gateway_log_group.arn
    format          = "[$context.requestId] ($context.identity.sourceIp) \"$context.httpMethod $cont
  }
}
```

Distributed Tracing with AWS X-Ray

AWS X-Ray offers a tracing perspective into the performance of your applications. It pinpoints how well your applications are operating and helps to identify bottlenecks. When following a request journey from Client –> API Gateway –> SQS –> Lambda –> S3, it's crucial to have such visibility. Integrating X-Ray with managed services like API Gateway and Lambda is straightforward.

Enable distributing tracing with X-Ray for your Lambda functions by adding the `tracing_config` element to the function resource definition:

```
resource "aws_lambda_function" "greeting_lambda" {
  # ... (other configurations)

  tracing_config {
    mode = "Active"
  }
}
```

Enable distributing tracing with X-Ray for your API Gateway by adding the `xray_tracing_enabled` element to the stage resource definition:

```
resource "aws_api_gateway_stage" "api_gateway_stage" {
  # ... (other configurations)

  xray_tracing_enabled = true
}
```

Make sure that the execution roles have permissions for publishing telemetry to X-Ray, e.g.

```
resource "aws_iam_policy" "lambda_xray_policy" {
  name        = "LambdaXRayPolicy"
  description = "Allows Lambda to send trace data to X-Ray."

  policy = jsonencode({
    Version = "2012-10-17",
    Statement = [
      {
        Action   = ["xray:PutTraceSegments", "xray:PutTelemetryRecords"],
        Effect   = "Allow",
        Resource = "*"
      }
    ]
  })
}

resource "aws_iam_role_policy_attachment" "lambda_xray_attach" {
  policy_arn = aws_iam_policy.lambda_xray_policy.arn
  role       = aws_iam_role.lambda_execution_role.name
}
```

Tagging Resources

Tagging resources in straightforward with Terraform - you simply attach your desired tags to the relevant resources. The advantage is that you can define a Terraform variable at the start of your `main.tf` file and then reference these values throughout your configurations for consistent tagging. This approach ensures uniformity while keeping your setup clean and organized.

```
resource "aws_lambda_function" "greeting_lambda" {
  # ... (other configurations)

  tags = {
    Environment = "Production"
    OwnerTeam   = "Engineering"
  }
}

resource "aws_api_gateway_rest_api" "greeting_api" {
  # ... (other configurations)

  tags = {
    Environment = "Production"
    OwnerTeam   = "Platform"
  }
}
```

Recap

In this chapter you've learned how to:

1. Set up logging for both Lambda and API Gateway, sending logs to CloudWatch for easy monitoring and debugging.
2. Integrate AWS X-Ray for both Lambda and API Gateway for in-depth insights into the application's execution and performance.
3. Institute a tagging strategy across your application resources for streamlined resource management and cost allocation.

Observability is a linchpin for successful and manageable modern infrastructures. Having insights into how components of your applications interact and perform under various conditions is invaluable in timely issue resolution, as well as cost and performance management.

Next Steps

In the previous chapters you've learned how to construct a scalable and comprehensive architecture, however all of the IaC configuration resided within a monolithic `main.tf` file. In following chapters you will learn how to refactor the monolithic `main.tf` into a collection of modular, reusable, and configurable Terraform modules.

8. Modularizing your IaC projects

In previous chapters, you've learned the foundational steps of constructing a serverless application IaC using a single `main.tf` file. In this chapter, you'll learn to elevate your IaC by transitioning from a basic one-file setup to a modular configuration. Your goal is to ensure scalability, maintainability, and the adoption of best practices that align with the demands of contemporary IT infrastructures.

In real-world scenarios, engineers would typically prefer a modular, decoupled approach to their IaC. They want to maintain the development flexibility, while ensuring operational efficiency. For example, it is common to separate resource definition and environment-specific configuration by segmenting your IaC project into different `.tf` files. Another common approach is to separate application logic code from IaC. This allows engineering teams to enhance agility across development, while enforcing cohesiveness throughout multiple environments and the product lifecycle.

Terraform modules

In order to support modular IaC definitions, Terraform provides support for Modules. Modules help to encapsulate resources for multiple related segments of your application. In previous chapters you've seen the `main.tf` file in the root working directory. This was an implicitly used *root module*. Modules can have input parameters, or variables, commonly defined in a `variables.tf` file, as well as outputs, commonly defined in an `outputs.tf` file, as you've seen in previous chapters.

`module-root-directory/`
- `variables.tf`
- `main.tf`
- `outputs.tf`

A module can include other modules, optionally passing down variable values. This is called using a child module. The syntax for including a child module is as follows. Outputs from child modules can be accessed from the parent module.

```
module "child_module_name" {
  source = "./child-module-directory"
  variable1 = value1
  variable2 = value2
  ...
}
```

Read more about Terraform Modules in the Terraform docs.

Directory structure

It is important to understand that there's no one-size-fits-all directory structure. Depending on factors like your existing Terraform practices or your anticipated deployment model, you might already have a preferred approach in your enterprise. When designing your project directory structure, it is important to understand not only *how* you're going to segment your files and modules, but also *why* you're going to organize it in a particular way. For simplicity, in this guide we'll organize Terraform modules by resource types, while decoupling source code into its own separate directory.

Below layout can serve as a good starting point for modularizing Terraform configurations and separating application source code into its own directory:

`project-root-directory/`

- `terraform /`
  - `main.tf`
  - `variables.tf`
  - `modules/`
  - `environments/`

- `lambda /`
  - `my-function-1/`
    - `index.mjs`

- The `terraform/` directory holds all Terraform IaC files. You put module definitions under `modules/` and different environment definitions under `environments/`.
- The `lambda/` directory contains your AWS Lambda functions. You would have a directory per function here, so you can add your code dependencies, such as `node_modules`, and other code related artifacts.

For reusability and clarity, encapsulate each resource type in its own module. Create individual folders under `modules/` for each module. It is a good idea to keep related resources like execution roles for Lambda functions in the same module as the Lambda function.

`terraform/`

- `modules/`
  - `apigateway/`
  - `lambdas/`
  - `sqs/`
  - `storage/`

- `environments/`

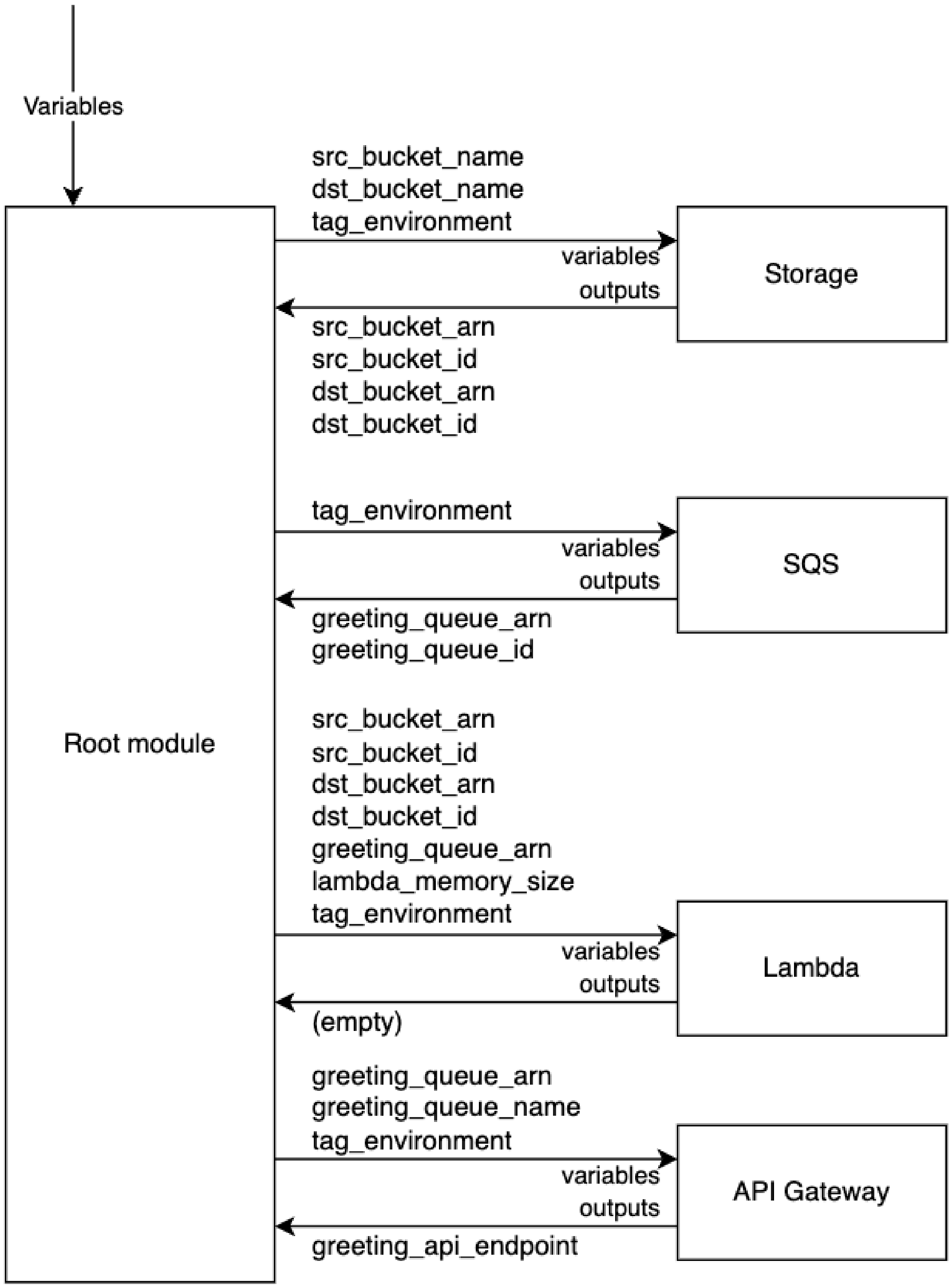
Each module should have its own `main.tf`, `variables.tf`, and `outputs.tf` files.

See below the outline of the `storage` module of the serverless application you've learned to build in previous chapters. It uses variables (`variables.tf`) to receive source and destination bucket names, as well as environment tag, creates required resources (`main.tf`), and outputs ARNs and IDs of created resources (`outputs.tf`). The internal logic is encapsulated within the module.

```
# variables.tf
variable "src_bucket_name" {
  type = string
}

variable "dst_bucket_name" {
  type = string
}

variable "tag_environment" {
  type = string
}
```

```
# main.tf
resource "aws_s3_bucket" "src_bucket" {
  bucket = var.src_bucket_name
  tags = {
    environment = var.tag_environment
  }
}



resource "aws_s3_bucket" "dst_bucket" {
  bucket = var.dst_bucket_name

  tags = {
    environment = var.tag_environment
  }
}



# outputs.tf
output "src_bucket_arn" {
  value = aws_s3_bucket.src_bucket.arn
}

output "src_bucket_id" {
  value = aws_s3_bucket.src_bucket.id
}

output "dst_bucket_arn" {
  value = aws_s3_bucket.dst_bucket.arn
}

output "dst_bucket_id" {
  value = aws_s3_bucket.dst_bucket.id
}
```

See below for a full visual representation of a modularized project structure.


Variables
src_bucket_name
dst_bucket_name
tag_environment
variables
outputs
Storage
src_bucket_arn
src_bucket_id
dst_bucket_arn
dst_bucket_id
tag_environment
variables
outputs
SQS
greeting_queue_arn
greeting_queue_id
Root module
src_bucket_arn
src_bucket_id
dst_bucket_arn
dst_bucket_id
greeting_queue_arn
lambda_memory_size
tag_environment
variables
outputs
Lambda
(empty)
greeting_queue_arn
greeting_queue_name
tag_environment
variables
outputs
API Gateway
greeting_api_endpoint

Environments

Differentiating resource configurations based on environments, such as development, staging, and production, enables organizations to maintain a clean separation of concerns. Each environment can contain its specific set of defaults, configurations, and resource allocations tailored to its purpose. This segregation ensures that experimental changes do not impact live systems, provides a controlled space for testing and validation, and allows for optimal resource utilization. Moreover, it simplifies the process of environment-specific configurations, making the infrastructure more modular and easier to manage, thus streamlining the overall deployment and scaling processes.

`terraform/`

- `modules/`
- `environments/`
    - `development/`
        - `terraform.tfvars`
    - `staging/`
        - `terraform.tfvars`
    - `production/`
        - `terraform.tfvars`

Each environment directory can have its own `terraform.tfvars` file, which contains variable values for that specific environment.

```
# environments/development/terraform.tfvars
environment        = "development"
lambda_memory_size = 256
```

```
# environments/production/terraform.tfvars
environment        = "production"
lambda_memory_size = 1024
```

When applying the Terraform configuration, you can either explicitly pass variable values, or specify which environment variables file to use.

```
terraform apply \
  -var="environment=production" \
  -var="lambda_memory_size=1024"
```

```
terraform apply \
  --var-file="environments/production/terraform.tfvars"
```

Externalizing configurations

While it is possible to store variable configuration values in a `.tfvars` file, you can go one step further and externalize them to a designated cloud service. AWS Systems Manager Parameter Store (SSM) provides centralized storage for configuration data. Externalizing the values to the SSM for environment differentiation allows to ensure a reinforced boundary separation. Each environment — whether it's development, staging, or production — has its distinct set of SSM parameters. As these parameters traverse their dedicated pipelines, the pipeline role specific to a particular environment is restricted to read only that environment's SSM parameters. This method not only enhances security and confines sensitive data but also streamlines configuration management. By tightly coupling roles with their respective environment's parameters, clear boundaries are maintained, preventing inadvertent data access or overlaps, and ensuring a coherent and safeguarded deployment process.

In order to create an externalized variable value for Terraform, you need to follow three steps

1. Create an SSM Parameter, for example

- Name = `/config/dev/resource-name`, value = `Development Resource`
- Name = `/config/prod/resource-name`, value = `Production Resource`

2. Retrieve SSM Parameters value, for example

```
# terraform/environments/dev/variables.tf
data "aws_ssm_parameter" "resource_name" {
  name = "/config/dev/resource-name"
}
```

```
# terraform/environments/prod/variables.tf
data "aws_ssm_parameter" "resource_name" {
  name = "/config/prod/resource-name"
}
```

```
variable "resource_name" {
  description = "The resource name, value is applied dynamically per environment"
  default     = data.aws_ssm_parameter.resource_name.value
}
```

3. Pass the populated variable to relevant modules:

```
# terraform/environments/dev/main.tf
module "some-module" {
  source = "../../modules/some-module"
  resource_name = var.resource_name
}
```

This approach ensures that the configurations in each environment fetch their specific parameters from SSM, and then those parameters are passed to the modules, which in turn apply them. This maintains a clean and modular project structure.

Conclusion

The deliberate separation of resources and environments in a directory structure ensures a clear and logical organization of resources, tailored specifically for each stage of the application's lifecycle. This modular approach guarantees that changes in one environment do not inadvertently impact another, fostering safer development and deployment practices.

Furthermore, the integration of AWS Systems Manager Parameter Store (SSM) into this structure amplifies the benefits. By centralizing configuration data within SSM, you eliminate hardcoded values, thereby enhancing security and maintainability. Additionally, by pulling environment-specific parameters from SSM, you ensure that each environment is uniquely configured, further strengthening the boundaries between them.

In essence, the combination of a well-organized environment-based folder structure with the dynamic, centralized parameter management offered by SSM results in a robust, scalable, and secure infrastructure setup. It paves the way for efficient development, testing, and deployment processes, ensuring that the right configurations are always applied to the right environments.

Recap

In this chapter you've learned how to:

1. Define a Terraform module
2. Organize your Terraform IaC into a collection of logical modules
3. Support multiple environments via `.tfvars` files or with the Systems Manager Parameter Store (SSM) service.

Next Steps

The next chapter will summarize this guide, as well as provide you with a collection of actionable next steps to continue your learning journey with Serverless and Terraform.

9. Summary and next steps

Hashicorp Terraform is an IaC tool that allows engineering teams to define and deploy their application environments using a declarative, human-readable syntax. Throughout this guide you've learned how to use Terraform for building serverless applications.

- How to add various resource types and build a decoupled serverless architecture
- How to implement synchronous APIs with API Gateway, and async processing with SQS message queues
- How to wire application components to work together while following the principle of least privileged IAM access
- How to test your applications locally with AWS SAM
- How to deploy your applications to AWS, and test them running in cloud
- How to add observability aspects, such as logging and tracing, to your applications
- How to elevate your Terraform templates by adding modules, supporting multiple environments, and separating infrastructure and application code.
- How to elevate your Terraform templates by adding modules, supporting for multiple environments, and separating infrastructure and application code.

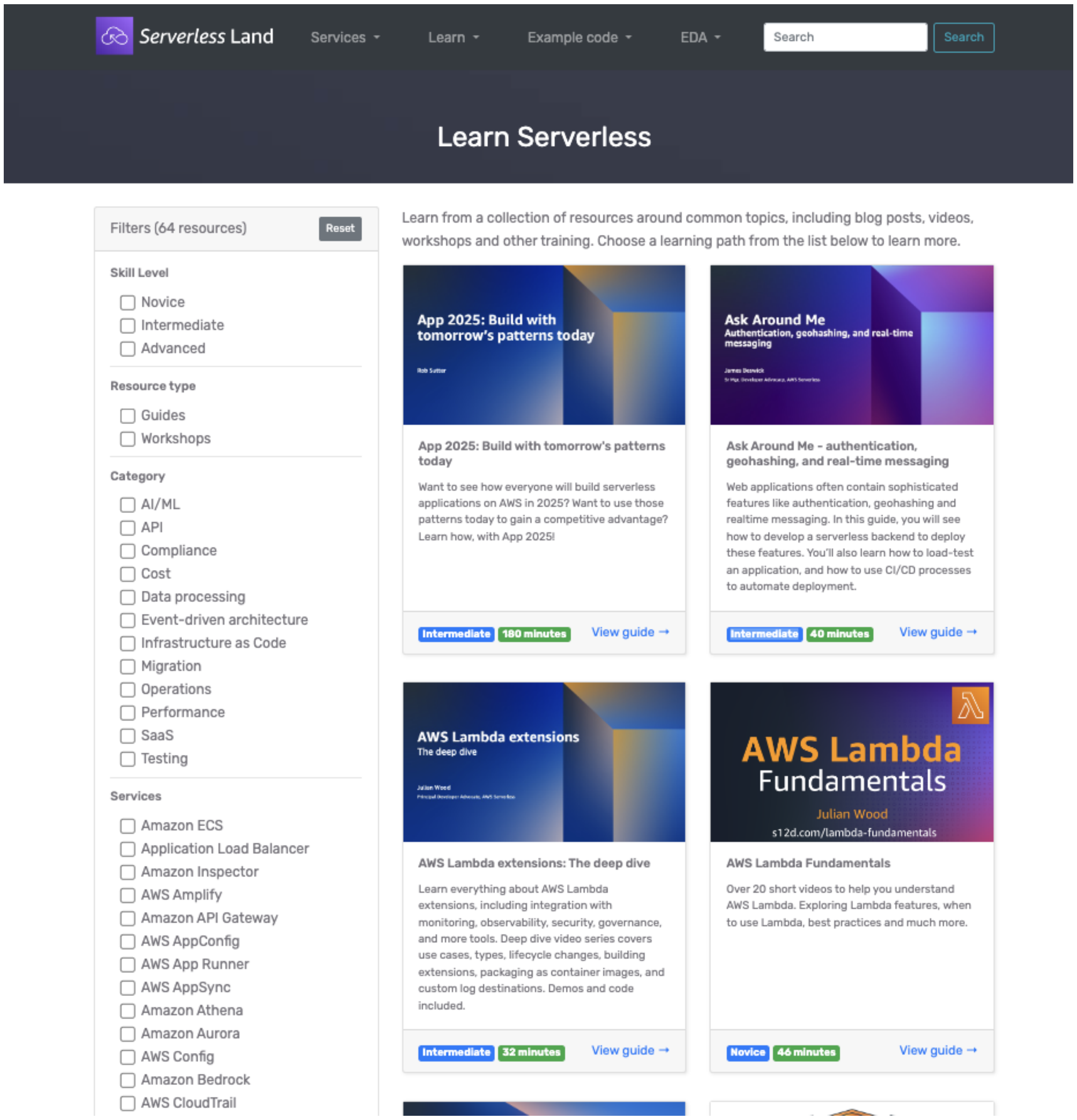
Next steps

Follow below links to continue your learning journey

- Getting Started with Serverless for Developers (guide)
- Implementing Governance in Depth for Serverless Applications (guide)
- Serverless Land - Serverless Patterns using Terraform
- Serverless Land - Learning guides and Workshops
- Using Terraform with AWS SAM | Serverless Office Hours (video)
- Building Serverless Applications with Terraform (workshop)
- Deploy serverless applications with AWS Lambda and API Gateway (tutorial)
- AWS Terraform Provider (documentation)
- Deploying AWS Step Functions state machines using Terraform (documentation)
- Best Practices for Writing Step Functions Terraform Projects (blog)
- Building AWS Step Functions with Terraform (workshop)
- Serverless Land - Step Functions Workflows using Terraform

Continue your learning

If you want to learn more about Serverless on AWS you can visit serverlessland.com. Serverless Land has hundreds of patterns, workflows, guides, workshops to explore.



Learn Serverless

Filters (64 resources) Reset

Skill Level
- ☐ Novice
- ☐ Intermediate
- ☐ Advanced

Resource type
- ☐ Guides
- ☐ Workshops

Category
- ☐ AI/ML
- ☐ API
- ☐ Compliance
- ☐ Cost
- ☐ Data processing
- ☐ Event-driven architecture
- ☐ Infrastructure as Code
- ☐ Migration
- ☐ Operations
- ☐ Performance
- ☐ SaaS
- ☐ Testing

Services
- ☐ Amazon ECS
- ☐ Application Load Balancer
- ☐ Amazon Inspector
- ☐ AWS Amplify
- ☐ Amazon API Gateway
- ☐ AWS AppConfig
- ☐ AWS App Runner
- ☐ AWS AppSync
- ☐ Amazon Athena
- ☐ Amazon Aurora
- ☐ AWS Config
- ☐ Amazon Bedrock
- ☐ AWS CloudTrail

Learn from a collection of resources around common topics, including blog posts, videos, workshops and other training. Choose a learning path from the list below to learn more.

App 2025: Build with tomorrow's patterns today

Want to see how everyone will build serverless applications on AWS in 2025? Want to use those patterns today to gain a competitive advantage? Learn how, with App 2025!

Intermediate 180 minutes View guide →

Ask Around Me - authentication, geohashing, and real-time messaging

Web applications often contain sophisticated features like authentication, geohashing and realtime messaging. In this guide, you will see how to develop a serverless backend to deploy these features. You'll also learn how to load-test an application, and how to use CI/CD processes to automate deployment.

Intermediate 40 minutes View guide →

AWS Lambda extensions: The deep dive

Learn everything about AWS Lambda extensions, including integration with monitoring, observability, security, governance, and more tools. Deep dive video series covers use cases, types, lifecycle changes, building extensions, packaging as container images, and custom log destinations. Demos and code included.

Intermediate 32 minutes View guide →

AWS Lambda Fundamentals

Over 20 short videos to help you understand AWS Lambda. Exploring Lambda features, when to use Lambda, best practices and much more.

Novice 46 minutes View guide →

Figure 1: Learn more on Serverlessland.com