**16**

**Hands-On Guide to Building an App in AWS**

In this chapter, you will combine many concepts you learned about in previous chapters to build a practical, e-commerce serverless architecture for a fictional online store called *AWSome Store*. After reading this chapter, you should be able to build your application using some or all of the components that you will review in this chapter.

For many of the services, you can plug and play and pick and choose the services that will be used in the application you develop. For example, your application may be more user-centric and may not require any asynchronous services. In this case, you can pull the asynchronous component out of your architecture. However, including some of the services in your architecture is highly advisable. For example, suppose your application is going to have a front end. In that case, you will want to ensure that you have an authentication component so that every user is authenticated before using any other application services.

In this chapter, you will first review the use case and build architecture upon your previous learning of *Domain Driven Architecture* in *Chapter 14*. You will build an architecture using AWS services and implement them using the AWS command line interface. You will cover the following topics:

* Understanding the use case
* Building the architecture for a fictional e-commerce website called *AWSome Store*
* Deciding on your programming language
* Implementing *AWSome Store* in AWS
* Optimizing with the Well-Architected Framework

Let’s start building in AWS now.

**An introduction to the use case**

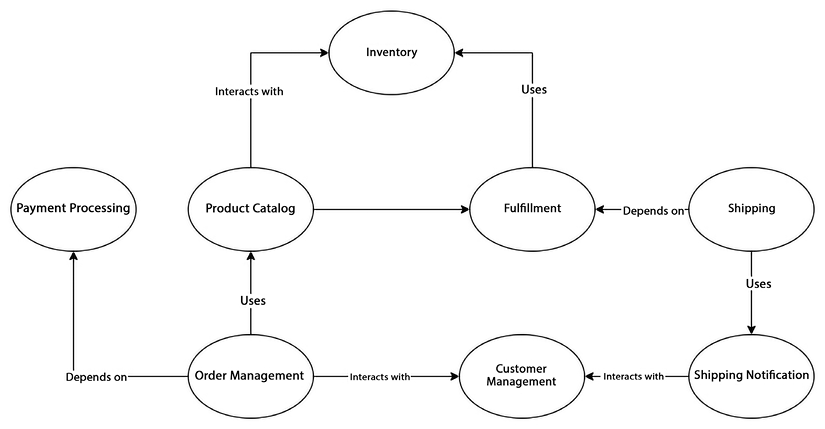
This chapter will allow you to combine and practice many concepts we have covered throughout this book. You’ll need to design the architecture of your system to scale without impacting application performance. You can use a serverless architecture, which involves building small, independent functions triggered by events, such as changes in a database or the arrival of a new message. This allows you to scale your system as needed, paying only for the resources you use.

To build upon your existing knowledge, let’s go into more detail about implementing domain-driven design for a retail e-commence application use case that you learned about in *Chapter 14*, *Microservice Architectures in AWS,* under the section *Domain-Driven Design*. Let’s take a trip down memory lane to understand the use case.

To make it a fun learning experience, we’ll give a name to the retail store we will use to learn AWS, let’s name it *AWSome Store*. Our imaginary online store provides AWS merchandise, such as stickers, books, water bottles, etc. You first need to identify the core business concepts, or entities, that make up the *AWSome Store* system to build **domain-driven design** (**DDD**). In the context of a retail e-commerce app, these entities include products, orders, customers, and payments. Next, you’ll need to identify the different contexts in which these entities exist. For example, the product context might include product details, availability, and pricing, while the order context might include information about order status, shipping, and billing.

Once you’ve identified the bounded contexts, you’ll need to define the relationships between the entities within each context. For example, an ordered aggregate might include an order, the customer who placed the order, and the purchased products. You may also identify services within each context that perform specific operations, such as calculating the total price of an order or processing a payment. Finally, you’ll need to identify domain events that trigger actions within the system, such as a new order being placed or a product becoming unavailable.

*Figure 16.1* shows a context map diagram for the *AWSome Store* for identified bounded contexts in the different business areas or contexts that make up the *AWSome Store* system, such as order management, payment processing, order fulfillment, shipping and customer management, etc.

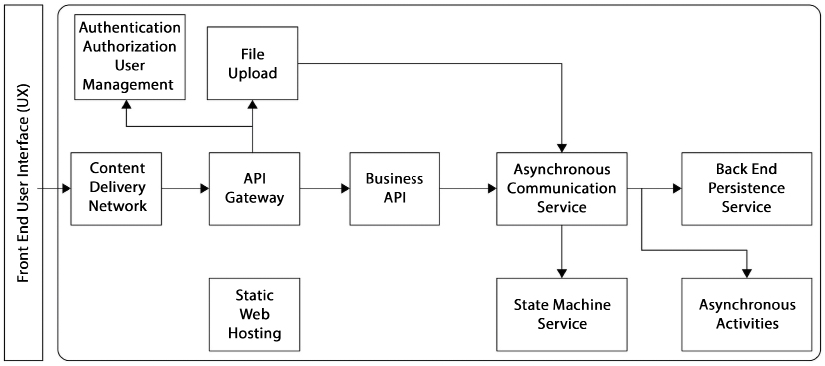


*Figure 16.1: Domain-driven design context map diagram for AWSome Store*

*Figure 16.1* defines the relationships between the different bounded contexts. For example, the payment processing context might be related to the order management context because products are ordered and sold. We have represented each bounded context as a separate oval on the diagram and we labeled each oval with the name of the context and the main entities within that context. The lines between the ovals represent the relationships between the bounded contexts and the type of relationship, such as *uses*, *depends on*, or *interacts with*. You can further refine the diagram as needed and review it with stakeholders to ensure that it accurately represents the relationships and dependencies between the bounded contexts.

By creating a context map diagram, you can understand the relationships between the different business areas of the *AWSome Store* e-commerce retail app, helping you make informed decisions about architecture, design, and implementation.

Let’s start looking at the design for our serverless microservice application. First, we’ll present the high-level architecture, and afterward, we’ll analyze each component or *domain* independently in detail:



*Figure 16.2: Serverless web-application architecture*

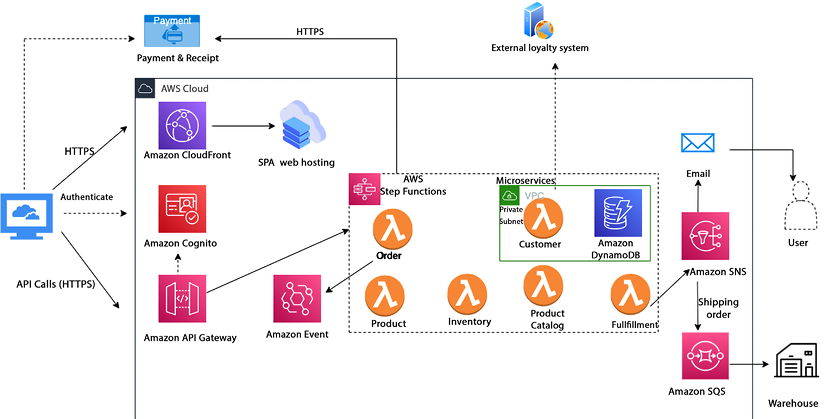
As you can see, the architecture provides services for static web hosting, business services, an asynchronous service, a state-machine service, and file uploads, among other components. In the following sections, we will look in detail into each of these components.

In *Figure 16.2*, each block represents delimited domains and technical functionality in many serverless implementations. Depending on the methodology used and whether you decide to use a microservice architecture, each of these blocks will represent one or more microservices. For example, the business API should be broken down into a series of microservices and not just one. You might have a microservice to handle *accounts* and another microservices to manage *companies*.

**Building architecture in AWS**

Now that you understand different domains and relationships, lets take a shot at building an architecture diagram using AWS services. AWS offers several services that you can use to build a robust, scalable, and resilient system. For example, you can use Amazon Lambda to build serverless functions, Amazon DynamoDB to store data, Amazon S3 to store files, and Amazon API Gateway to create APIs. Also, implementing proper error handling and retry logic is crucial for building a resilient and robust system with AWS Lambda.

To allow for the quick implementation of new features, you’ll need to implement a CI/CD pipeline that automatically builds, tests, and deploys new code to production. Let’s look at *AWSome Store*’s proposed architecture using AWS cloud-native services:



*Figure 16.3: AWS cloud-native architecture to build AWSome Store*

As shown in *Figure 16.3*, AWS provides various cloud-native serverless technologies to implement a scalable and resilient e-commerce retail app, such as AWS Lambda, Amazon API Gateway, Amazon DynamoDB, Amazon S3, and Amazon EventBridge. These services could be used to implement a cloud-native serverless architecture for the following key components of the *AWSome Store*:

* **Front end**: The front end of the *AWSome Store* could be built using a JavaScript framework such as React and hosted on Amazon S3. Amazon CloudFront is used as a **Content Delivery Network** (**CDN**) to make sure website content is available to global users close to their location.
* **Authentication and Authorization**: Amazon Cognito is used to authenticate end customers and authorize them to use the application for profile creation and placing orders.
* **API Gateway**: Amazon API Gateway is used to create RESTful APIs for the front end to access the back-end services and to securely call them. API Gateway provides several features to help protect your API calls, including authentication and authorization mechanisms such as OAuth 2.0, API keys, and Amazon Cognito user pools. You can also use API Gateway to implement throttling, rate limiting, and caching to help prevent abuse and improve performance.
* **Lambda functions**: AWS Lambda is used to build the back-end functionality of the *AWSome Store*. The application logic could be broken down into multiple Lambda functions, each handling a specific aspect of the app, such as product catalog management, order management, and payment processing.
* **DynamoDB**: Amazon DynamoDB is the primary data store for the *AWSome Store*, providing a scalable and managed NoSQL database for storing customer information, the product catalog, and order information.
* **EventBridge**: Amazon EventBridge is used to build event-driven architectures, enabling the application to respond to events such as new orders and payment transactions.
* **S3**: Amazon S3 could store and retrieve product images and other static assets such as a PDF file for a product manual.

Other AWS services can be used in conjunction with the above key services to ensure security, build deployment pipelines, carry out log monitoring, raise system alerts, etc. For example, AWS IAM helps you control access to AWS resources. You can use IAM to create and manage user accounts, roles, and permissions, which can help you ensure that only authorized users have access to your serverless app services. You can use CloudFormation to define and manage your infrastructure as code. AWS CloudTrail records API calls and events for your AWS account. You can use CloudTrail to monitor activity in your serverless app services and to troubleshoot issues by reviewing the history of API calls and events. You can also use CloudWatch to collect and track metrics, collect and monitor log files, and set alarms. You can use AWS Config to provide a detailed inventory of your AWS resources, including configurations and relationships between resources.

You will learn about them as we move forward with the implementation details.

**Deciding which is the best language**

An important question is what language is best for you, your project, and your company. As a general rule of thumb, if you have a greenfield project where you have a wide choice of languages to pick from, it may be best to go with one of the newer languages, and it also makes sense to pick a popular language such as Golang.

This choice is not always straightforward and clear cut. A language’s popularity is not static. As an example, Perl was one of the most popular languages in the 1990s, but its popularity has severely waned. So, it’s not enough to consider the popularity of a language, but also how fast it’s growing or fading away. Similarly, JavaScript was the most popular, but it was overtaken by Python in 2022. You can refer to this article published by GitHub to see the current trends in programming language popularity – <https://octoverse.github.com/2022/top-programming-languages>.

If a language is popular, you will easily find resources for your project. Some other considerations to keep in mind are shown in the following list:

* **Compiled versus interpreted**: If you don’t expect your project to become the next Airbnb and know that the number of users or the workload will be capped, you might be better off using an interpreted language rather than a compiled language. Interpreted languages allow you to prototype and ramp up your application quickly. Being able to fail fast is key to succeeding fast eventually. Fast development cycles allow us to test our ideas quickly and discard the ones that don’t perform as expected. Usually, the development life cycle is quicker with an interpreted language because the code doesn’t need to be compiled every time there is a code change. A compiled language may be better if your application has strict security requirements.
* **Problem domain**: If you have a small project and you are not working in the context of a company, the better choice may hinge on what other people have already done. You may be a highly experienced Java developer, but perhaps someone has already solved 90% of the requirements you are trying to cover. In this case, you may be better off teaching yourself Python to save replicating a lot of work.
* **Staff availability**: After researching, you may conclude that Ruby is the best language and has everything you need. But if you expect the application to require a sizable team to take it to completion and Ruby developers are in short supply (and therefore command high rates), it may be best to settle for second best and not be a language purist.

Regardless of your language selection, if you design your application correctly and leverage the advantages of a microservice architecture, you will be well on your way to a successful implementation. The combined microservice architecture with serverless deployment in AWS or a similar environment has been the recipe for many recent hits, some so successful that billion-dollar companies have been created around these products and services.

It is time to start putting some components together using the microservice architecture we learned about in *Chapter 14*, *Microservice Architectures in AWS*.

**Setting up services**

To initiate the development of an application, the initial step involves establishing the essential infrastructure services. As you have chosen to utilize AWS services extensively, these services can be configured via various means, such as the AWS Console, AWS**Cloud Development Kit** (**CDK**), AWS CloudFormation, the **Command Line Interface** (**CLI**), or through third-party tools like Chef, Puppet, Ansible, Terraform, etc.

In this section, the main emphasis is on how to use these services while building cloud-native applications rather than constructing the infrastructure itself. Therefore, this chapter will focus on the process of setting up services utilizing CLI. You will go through the following steps to implement *AWSome Store* in AWS:

1. Set up an AWS account
2. Install the AWS CLI
3. Set up IAM users, roles, and groups
4. Create the AWS infrastructure
5. Implement authentication and authorization for end customers
6. Define database attributes
7. Write the code for *AWSome Store* using Lambda functions
8. Deploy and test your code
9. Logging and monitoring

**Set up an AWS account with a billing alert**

The first step is to set up an AWS account. By now, you have likely already set up an account, but here are the steps to set up an AWS account in case you haven’t:

1. Open a web browser and navigate to the AWS home page at <https://aws.amazon.com/>.
2. Click on the **Create an AWS Account** button located in the top-right corner of the page.
3. Fill in your account information, including your email address, password, and account name.
4. Provide your contact information, such as your name, company name, and phone number.
5. Enter your payment information, including your credit card details.
6. Choose a support plan that meets your needs, whether it’s Basic, Developer, Business, or Enterprise.
7. Read and accept the AWS Customer Agreement and the AWS Service Terms.
8. Click the **Create Account and Continue** button.
9. AWS will send a verification code to your email address or phone number. Enter the code to verify your account.
10. Once your account is verified, you can start using AWS services.
11. To find your AWS account number, click on your name or account ID in the navigation bar at the top-right corner of the console. On the **My Account** page, you will see your 12-digit AWS account number displayed in the **Account Identifiers** section under **Account Number**.

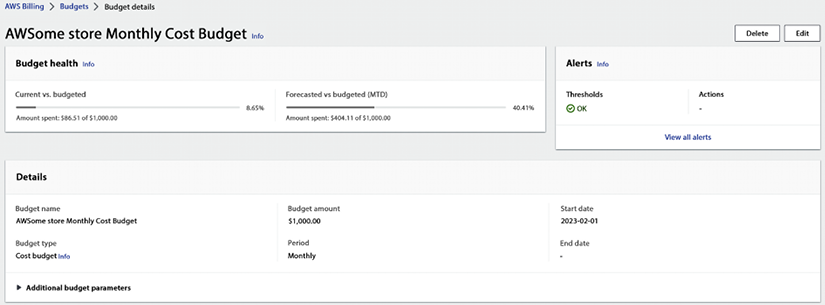
After completing the verification process, you will have access to the AWS Management Console and can start using AWS services. It’s important to note that you may need to provide additional information or documentation to fully activate your AWS account, especially if you plan to use AWS services that involve payment or access to sensitive data.

Additionally, it’s a good idea to set up billing alerts to ensure that you are aware of the costs incurred while using AWS services. The costs can add up fast if you are not careful.

Here are the steps to set up billing alerts in AWS:

1. In the AWS console, navigate to **AWS Budget** by clicking on the **Services** drop-down menu and selecting **AWS Cost Management.**
2. On the **Budgets** screen, click on the **Create budget** button.
3. You can use a template to create a budget, for example, **Monthly cost budget**.
4. Provide a name for your budget, the budgeted amount, and the email recipients and select the **Create Budget** button to create the budget and complete the setup process.

Your final billing alert will look like *Figure 16.4*.



*Figure 16.4: AWS billing alert dashboard*

After setting up a budget, you will receive alerts via email when the budget threshold is exceeded. It’s a good idea to regularly review your budgets and make adjustments as needed to ensure that you are always aware of your AWS spending.

As you have created an AWS account and set up your billing alert, now install CLI to create the rest of the services in AWS.

**Install the AWS Command Line Interface (CLI)**

The AWS CLI is available for Windows, macOS, and Linux. You can install the AWS CLI using the bundled installer, pip, or Homebrew. You can download the CLI installer and get started with AWS CLI by visiting the AWS user docs here – <https://aws.amazon.com/cli/>.

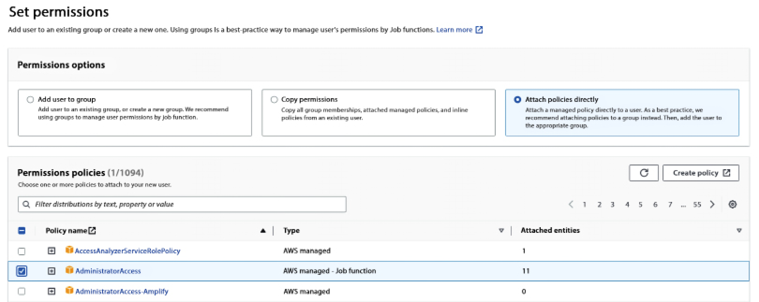
The root account has unrestricted access to all AWS services and resources in the account, which makes it a valuable target for attackers. If the root account is compromised, an attacker could gain complete control of your AWS resources and data, potentially causing significant harm to your business.

Creating an admin account in AWS is a best practice. It is important to have separate accounts with appropriate permissions to help ensure the security of your AWS resources and data. Creating an admin account with limited permissions reduces the risk of unauthorized access and reduces the impact of a potential security breach. Admin accounts can be given permissions to manage specific AWS services and resources, without granting them the full access of the root account.

By creating an admin account, you can better control and monitor the use of AWS services and resources and reduce the risk of security breaches or accidental changes that can impact the availability, integrity, and confidentiality of your data.

To configure the CLI, you need an IAM user with an access key ID and secret access key, so let’s create an admin user by following the below steps:

1. Log in to the AWS Management Console and go to the IAM dashboard.
2. Click on **Users** in the navigation menu.
3. Click on the **Add user** button.
4. Enter a username for the new user; in this case you can give the name *store admin*, and click the **Next** button to land on the **Set Permissions** screen.
5. Select the option **Attach policies directly** and tick the **AdministratorAccess** checkbox as shown below. To keep things simple, we have defined an admin user. As a best practice, always follow least-privilege permissions; when you set permissions with IAM policies, only grant the permissions required to perform a task. You can learn more here: <https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html>.



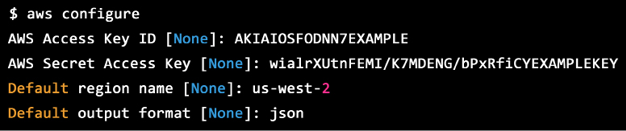
*Figure 16.5: AWS IAM policy attachment*

1. Review the user details and click on the **Create user** button.
2. Now go to the newly created *store admin* user by navigating to **Users**.
3. Select the **Security credentials** tab and click on **Create access key** in the **Access Key** section.
4. Retrieve the CSV file containing the access key ID and secret access key and ensure that it is stored securely, as it will only be accessible once and cannot be retrieved again later.

Now the new user has been created and has administrative permissions. You can now use the access key ID and secret access key to access AWS services using the CLI programmatically.

To configure the CLI, you need to run the aws configure command and provide your AWS access key ID, secret access key, default region name, and output format.

Here is an example of the configuration process:



*Figure 16.6: Sample configuration code*

The AWS CLI will prompt you for the following information:

* **AWS Access Key ID**: Your AWS access key ID allows you to access AWS services.
* **AWS Secret Access Key**: Your secret access key provides secure access to your AWS account.
* **Default region name**: The region where your AWS resources will be created. For example, us-west-2 for Oregon or us-east-1 for North Virginia.
* **Default output format**: The output from the AWS CLI commands. For example, JSON or text.

It’s important to note that you will need to have an AWS account and the necessary permissions to perform actions using the AWS CLI. When creating an IAM user, role, or group in AWS, it’s important to follow the principle of **zero trust**. This means that by default, you should assume that any user or entity is untrusted until proven otherwise. You don’t want anyone to have admin access so let’s create some IAM credentials using the CLI.

**Set up IAM users, roles, and groups**

Following the principle of **zero trust**, let’s create a dev user with read/write access to S3, Lambda, and API Gateway using the AWS CLI. You can follow these steps:

Open the command line or terminal on your system and run the following command to create a dev user:

aws iam create-user --user-name dev-user

Next, create an IAM policy for read/write access to S3:

aws iam create-policy \

--policy-name S3-ReadWriteAccess \

--policy-document file://s3-readwrite-policy.json

Where the s3-readwrite-policy.json file contains the following policy:

**Note**: The placeholder text <bucket-name> should be replaced with a unique name of your choice for the S3 bucket. For instance, you could use a name like aws-for-sa-book-s3-app. It’s important to ensure that the S3 bucket name is available and not already in use by another AWS account. Additionally, it’s recommended to follow AWS naming conventions for S3 bucket names, which include using lowercase letters, numbers, and hyphens, and not using underscores or uppercase letters.

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

"s3:ListBucket",

"s3:GetObject",

"s3:PutObject"

],

"Resource": [

"arn:aws:s3:::<bucket-name>/\*"

]

}

]

}

Similarly, create a policy for read/write access to Lambda:

aws iam create-policy \

--policy-name Lambda-ReadWriteAccess \

--policy-document file://lambda-readwrite-policy.json

Where the lambda-readwrite-policy.json file contains the following policy:

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

"lambda:ListFunctions",

"lambda:GetFunction",

"lambda:CreateFunction",

"lambda:UpdateFunctionCode",

"lambda:DeleteFunction"

],

"Resource": "\*"

}

]

}

And similarly, create a policy for read/write access to the API Gateway:

aws iam create-policy \

--policy-name API-Gateway-ReadWriteAccess \

--policy-document file://apigateway-readwrite-policy.json

Where the apigateway-readwrite-policy.json file contains the following policy:

**Note**: When creating an API gateway in AWS, you need to replace the placeholders <region-id> and <api-id> with the actual values for your environment. The <region-id> should be replaced with the AWS region identifier where you want to create the API gateway. For example, us-east-1 for the US East (N. Virginia) region.

The <api-id> should be replaced with a unique identifier for your API Gateway. This identifier is used to generate the API Gateway endpoint URL, so it’s important to choose a unique name that reflects the purpose of your API. For example, my-api-gateway or ecommerce-api. Make sure to choose appropriate values for both placeholders, as they will affect the configuration and functionality of your API Gateway.

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

"apigateway:GET",

"apigateway:POST",

"apigateway:PUT",

"apigateway:DELETE",

"apigateway:PATCH"

],

"Resource": "arn:aws:apigateway:<region-id>:<api-id>/\*"

}

]

}

Attach the policies to the dev user:

**Note**: When you create a new AWS account, it is assigned a unique 12-digit account number. When executing commands in AWS that require specifying the account number, you will need to replace the placeholder text with the actual account number for your account. For example, if a command requires specifying the account number as <account-number>, you should replace this placeholder with your actual 12-digit account number, like 123456789012. Using the correct account number is important to ensure that the command is executed in the correct AWS account, especially if you have multiple AWS accounts.

aws iam attach-user-policy --user-name dev-user --policy-arn arn:aws:iam::<account-id>:policy/S3-ReadWriteAccess

aws iam attach-user-policy --user-name dev-user --policy-arn arn:aws:iam::<account-id>:policy/lambda-ReadWriteAccess

aws iam attach-user-policy --user-name dev-user --policy-arn arn:aws:iam::<account-id>:policy/apigateway-ReadWriteAccess

Finally, verify the policies have been attached to the dev user:

aws iam list-attached-user-policies --user-name dev-user

Similarly, you should create policies for other users as needed. You can use a policy generator to build the right policy for each user by referring to this link – <https://awspolicygen.s3.amazonaws.com/policygen.html>.

Security is most important for your application, so make sure to use the below best practices when creating IAM entities with zero-trust principles in mind:

* **Least privilege**: Assign the minimum permissions necessary to accomplish a task. For example, if a user only needs to access a specific S3 bucket, only grant them access.
* **Role-based access control**: Use IAM roles instead of long-term access keys. Roles are more secure because they are limited to a specific set of permissions and can be revoked at any time.
* **Multi-factor authentication**: Require **multi-factor authentication** (**MFA**) for all IAM entities, including root, users, roles, and groups. MFA provides an extra layer of security and helps to prevent unauthorized access.
* **Use policies and conditions**: Define policies and conditions that limit access to AWS resources based on factors such as the time of day, IP address, and user agent.
* **Monitor and audit**: Regularly monitor and audit your AWS environment to detect potential security issues and ensure that users are only accessing resources as intended.
* **Use managed policies**: Use managed policies instead of custom policies whenever possible. Managed policies are pre-built policies reviewed and approved by AWS security experts.
* **Regularly rotate credentials**: Regularly rotate access keys, passwords, and other credentials to ensure that only authorized users can access your AWS resources.

By following these best practices, you can ensure that your AWS environment is secure and that only authorized users can access your resources.

Let’s move on to the next step of creating cloud infrastructure for architecture.

**Create the AWS infrastructure**

After creating IAM credentials, let’s create the app infrastructure. For simplicity and to explain the concept, you will use the CLI to bring-up various AWS services; however, as a best practice you should choose the route of writing a CloudFormation template and using that to bring up your AWS infrastructure. Here is an example of how to create an API Gateway, S3, DynamoDB, CloudFront, and Lambda instance using the AWS CLI:

Create an S3 bucket:

aws s3 mb s3://<bucket-name>

**Note**: The placeholder text <bucket-name> should be replaced with a unique name of your choice for the S3 bucket.

Create a DynamoDB table:

aws dynamodb create-table --table-name awesome-store-table --attribute-definitions AttributeName=id,AttributeType=S --key-schema AttributeName=id,KeyType=HASH --provisioned-throughput ReadCapacityUnits=1,WriteCapacityUnits=1

Create a CloudFront distribution:

aws cloudfront create-distribution --distribution-config file://cloudfront-config.json

Here is some example code for a CloudFront configuration file in the JSON format. The file cloudfront-config.json should contain the CloudFront distribution configuration:

{

"Comment": "My CloudFront Distribution Configuration",

"Logging": {

"Bucket": "<my-logs-bucket>",

"IncludeCookies": true,

"Prefix": "my-cloudfront-logs/"

},

"Origins": {

"Quantity": 1,

"Items": [

{

"Id": "my-origin",

"DomainName": "<my-origin-domain>",

"CustomOriginConfig": {

"HTTPPort": 80,

"HTTPSPort": 443,

"OriginProtocolPolicy": "https-only",

"OriginSslProtocols": {

"Quantity": 1,

"Items": ["TLSv1.2"]

}

}

}

]

},

"DefaultCacheBehavior": {

"TargetOriginId": "my-origin",

"ForwardedValues": {

"QueryString": false,

"Cookies": {

"Forward": "none"

}

},

"TrustedSigners": {

"Enabled": false,

"Quantity": 0

},

"ViewerProtocolPolicy": "redirect-to-https",

"MinTTL": 0,

"MaxTTL": 86400,

"DefaultTTL": 3600

},

"Enabled": true,

"PriceClass": "PriceClass\_All",

"DefaultRootObject": "index.html",

"Aliases": {

"Quantity": 2,

"Items": [

"<www.mydomain.com>",

"<mydomain.com>"

]

},

"ViewerCertificate": {

"ACMCertificateArn": "<my-acm-certificate-arn>",

"SSLSupportMethod": "sni-only"

}

}

This configuration file defines a CloudFront distribution that uses one origin, enables logging to an S3 bucket, and specifies cache behavior settings. Note that some values, such as <my-logs-bucket> and <my-origin-domain>, need to be replaced with actual values specific to your environment. For example, <my-logs-bucket> should be replaced with the name of an existing S3 bucket where CloudFront can store log files, such as my-cloudfront-logs-bucket. In the same line, <my-origin-domain> should be replaced with the domain name of an existing origin server for your CloudFront distribution, such as www.example.com.

When creating an API Gateway API and a Lambda function in AWS, you can use any valid name of your choice for these resources, as long as the name is unique within your AWS account and follows the naming rules and restrictions for AWS resource names. In the examples below, awesome-store-api and awesome-store-lambda are just example names that could be replaced with other names of your choosing. Just make sure to use the same names consistently throughout your code and configurations when referring to these resources.

Create an API Gateway REST API:

aws apigateway create-rest-api --name awesome-store-api --description "Awesome Store API"

Create a Lambda function:

aws lambda create-function --function-name awesome-store-lambda --runtime nodejs12.x --handler index.handler --zip-file fileb://lambda.zip

**Note**: The lambda.zip file typically contains the code for the AWS Lambda function that you want to deploy. In the example command you provided, the code in lambda.zip is for a Node.js 12.x runtime environment. However, AWS Lambda supports a variety of other programming languages and runtime environments, including Python, Java, C#, Go, and Ruby, among others. To deploy code for a different runtime environment, you would need to make sure that the code is compatible with that specific runtime environment, and then package it into a new ZIP file with the appropriate file extension for that language (such as .py for Python or .jar for Java). Once you have the new ZIP file with your code for the desired runtime environment, you can use a similar command to the one you provided, but specify the appropriate runtime flag and other parameters as needed for the specific language and environment.

**Create the AWS infrastructure using CloudFormation**

Above was a basic example of creating these AWS resources using the CLI. Here’s an example CloudFormation template to deploy the resources needed for an *AWSome Store* e-commerce website:

---

AWSTemplateFormatVersion: '2010-09-09'

Description: CloudFormation template to deploy resources needed for an AWSome Store e-commerce website

Resources:

S3Bucket:

Type: AWS::S3::Bucket

Properties:

BucketName: awesome-store-bucket

WebsiteConfiguration:

IndexDocument: index.html

DynamoDBTable:

Type: AWS::DynamoDB::Table

Properties:

TableName: awesome-store-table

AttributeDefinitions:

- AttributeName: id

AttributeType: S

KeySchema:

- AttributeName: id

KeyType: HASH

ProvisionedThroughput:

ReadCapacityUnits: 1

WriteCapacityUnits: 1

CloudFrontDistribution:

Type: AWS::CloudFront::Distribution

Properties:

DistributionConfig:

Origins:

- Id: awesome-store-origin

DomainName: !Join ['.', [!Ref S3Bucket, 's3.amazonaws.com']]

S3OriginConfig:

OriginAccessIdentity: !Join ['/', ['origin-access-identity/cloudfront', !Ref AWS::AccountId]]

DefaultCacheBehavior:

TargetOriginId: awesome-store-origin

ViewerProtocolPolicy: redirect-to-https

Enabled: true

APIGateway:

Type: AWS::ApiGateway::RestApi

Properties:

Name: awesome-store-api

Description: Awesome Store API

LambdaFunction:

Type: AWS::Lambda::Function

Properties:

FunctionName: awesome-store-lambda

Runtime: nodejs12.x

Handler: index.handler

Code:

S3Bucket: !Ref S3Bucket

S3Key: lambda.zip

The above is just an example; you may need to modify it to meet your specific requirements. Additionally, you may need to add additional resources, such as IAM roles and policies, to implement the *AWSome Store* e-commerce website fully. You will learn more about writing Lambda code and putting it in lambda.zip later in this chapter. For the time being, you can copy the above code in awsome-store-app.yaml and run the following command to deploy the CloudFormation template:

aws cloudformation create-stack --stack-name <stack-name> --template-body file://<template-file>

Replace <stack-name> with the desired name of your CloudFormation stack and <template-file> with the name of your CloudFormation template file, for example:

aws cloudformation create-stack --stack-name awsome-store --template-body file://awsome-store-app.yaml

To monitor the progress of the CloudFormation stack creation, you can use the following command:

aws cloudformation describe-stacks --stack-name <stack-name>

Once the CloudFormation stack is created successfully, it will return a status of CREATE\_COMPLETE.

**Creating an EventBridge instance and a queue**

You can use AWS EventBridge and SQS to build loosely coupled architecture and send asynchronous messages like emails for customer notifications. Here are the high-level steps to set up AWS EventBridge and SQS for the *AWSome Store* using the AWS CLI:

1. **Create an SQS queue**:

You can use the following AWS CLI command to create a new SQS queue:

aws sqs create-queue --queue-name awsome-store-queue

1. **Get the ARN of the SQS queue**:

You can use the following AWS CLI command to get the ARN of the SQS queue:

aws sqs get-queue-attributes --queue-url <queue-url> --attribute-names QueueArn

**Note**: To use the command above, you need to replace <queue-url> with the actual URL of the SQS queue for which you want to retrieve the attribute. For example, if the URL of your SQS queue is https://sqs.us-west-2.amazonaws.com/123456789012/my-queue, you would replace <queue-url> with that URL, like this:

aws sqs get-queue-attributes --queue-url https://sqs.us-west-2.amazonaws.com/123456789012/my-queue --attribute-names QueueArn

This command would retrieve the QueueArn attribute for the my-queue SQS queue in the us-west-2 region.

1. **Create a new AWS EventBridge rule**:

You can use the following AWS CLI command to create a new AWS EventBridge rule using QueueArn retrieved from the previous command:

aws events put-rule --name awsome-store-rule --event-pattern '{"source":["aws.sqs"],"detail-type":["SQS Message Received"],"detail":{"queue":["<queue-arn>"]}}'

1. **Add a target to the EventBridge rule**:

You can use the following AWS CLI command to add a target to the EventBridge rule:

aws events put-targets --rule awsome-store-rule --targets Id=1,Arn=<lambda-function-arn>

The aws events put-targets command adds one or more targets to an Amazon EventBridge rule, specified by the --rule parameter. In this case, the target being added is an AWS Lambda function, specified by its ARN using the --targets parameter. To use this command, you need to replace awsome-store-rule with the name of the EventBridge rule to which you want to add the target. You also need to replace <lambda-function-arn> with the ARN of the AWS Lambda function that you want to use as the target.

For example, if you have an EventBridge rule called my-event-rule and an AWS Lambda function with an ARN of arn:aws:lambda:us-west-2:123456789012:function:my-lambda-function, you would replace awsome-store-rule with my-event-rule, and replace <lambda-function-arn> with arn:aws:lambda:us-west-2:123456789012:function:my-lambda-function, like this:

aws events put-targets --rule my-event-rule --targets Id=1,Arn=arn:aws:lambda:us-west-2:123456789012:function:my-lambda-function

This command would add the my-lambda-function AWS Lambda function as a target to the my-event-rule EventBridge rule in the us-west-2 region.

Next, send a message to the SQS queue. You can use the following AWS CLI command to send a message to the SQS queue by using the queue URL you captured earlier:

aws sqs send-message --queue-url <queue-url> --message-body "Hello, this is a test message."

Once these steps are completed, you have successfully set up AWS EventBridge and SQS for the *AWSome Store* using the AWS CLI. You will learn about best practices to set up AWS infrastructure later, in the Optimization with Well-Architected Review section of this chapter.

**Implement authentication and authorization for end users**

Earlier you set up IAM credentials for the internal development team, but what about end users who will access your *AWSome Store* and build their profile? You need to give them a way to create their account securely and give them the required access. You can use Amazon Cognito, to simplify user authentication and authorization for cloud-based applications. It can help you with the following scenarios:

* **User sign-up and sign-in**: Cognito provides secure user authentication, with features such as multi-factor authentication, forgot password, and social identity sign-up.
* **Mobile and web app authentication**: Cognito supports users’ authentication in mobile and web applications and integrates with the AWS Mobile SDK.
* **Unauthenticated and authenticated access to APIs**: Cognito integrates with Amazon API Gateway and AWS AppSync to provide authorization for API access.
* **User data storage**: Cognito provides user profile storage, which can be used to store user information such as preferences, custom data, and more.

Here is an example of how you can use the AWS CLI to implement authentication and authorization for end users for the *AWSome Store* website using Amazon Cognito:

1. Create a Cognito user pool:
2. aws cognito-idp create-user-pool --pool-name awsomestore-pool

The aws cognito-idp create-user-pool command creates a new Amazon Cognito user pool with the specified pool name, using the default settings. To use this command, you need to replace awsomestore-pool with the name that you want to give to your user pool. For example, if you want to create a user pool called my-user-pool, you would replace awsomestore-pool with my-user-pool, like this:

aws cognito-idp create-user-pool --pool-name my-user-pool

This command would create a new Amazon Cognito user pool with the name my-user-pool. You can customize the settings of your user pool using additional parameters and options with this command.

1. Next, create a group for the user pool:
2. aws cognito-idp create-group --user-pool-id awsomestore-pool-id --group-name awsomestore-group

The aws cognito-idp create-group command creates a new group in the specified Amazon Cognito user pool with the specified group name. To use this command, you need to replace awsomestore-pool-id with the ID of the Amazon Cognito user pool in which you want to create the group. You also need to replace awsomestore-group with the name of the group that you want to create.

For example, if you have an Amazon Cognito user pool with an ID of us-west-2\_abc123xyz, and you want to create a new group called my-group, you would replace awsomestore-pool-id with us-west-2\_abc123xyz, and replace awsomestore-group with my-group, like this:

aws cognito-idp create-group --user-pool-id us-west-2\_abc123xyz --group-name my-group

This command would create a new group called my-group in the us-west-2\_abc123xyz Amazon Cognito user pool.

1. Next, attach policies to the group:
2. aws cognito-idp add-user-to-group --user-pool-id awsomestore-pool-id --username <username> --group-name awsomestore-group

The aws cognito-idp add-user-to-group command adds a user to a specified group in an Amazon Cognito user pool. To use this command, you need to replace awsomestore-pool-id with the ID of the Amazon Cognito user pool in which the group exists.

You also need to replace <username> with the username of the user you want to add to the group. Finally, you need to replace awsomestore-group with the name of the group to which you want to add the user.

For example, if you have an Amazon Cognito user pool with an ID of us-west-2\_abc123xyz, and you want to add a user with the username myuser to a group called my-group, you would replace awsomestore-pool-id with us-west-2\_abc123xyz, replace <username> with myuser, and replace awsomestore-group with my-group, like this:

aws cognito-idp add-user-to-group --user-pool-id us-west-2\_abc123xyz --username myuser --group-name my-group

This command would add the user with the username myuser to the my-group group in the us-west-2\_abc123xyz Amazon Cognito user pool.

1. Next, create a Cognito user pool client:
2. aws cognito-idp create-user-pool-client --user-pool-id awsomestore-pool-id --client-name awsomestore-client

The aws cognito-idp create-user-pool-client command creates a new app client in an Amazon Cognito user pool. To use this command, you need to replace awsomestore-pool-id with the ID of the Amazon Cognito user pool in which you want to create the app client. You also need to replace awsomestore-client with a name of your choice for the app client. For example, if you have an Amazon Cognito user pool with an ID of us-west-2\_abc123xyz, and you want to create a new app client called my-app-client, you would run the following command:

aws cognito-idp create-user-pool-client --user-pool-id us-west-2\_abc123xyz --client-name my-app-client

This command would create a new app client called my-app-client in the us-west-2\_abc123xyz Amazon Cognito user pool. The command will return the newly created app client’s ClientId value, which will be required to authenticate the users.

1. Next, create a Cognito identity pool:
2. aws cognito-identity create-identity-pool --identity-pool-name awsomestore-identity-pool --allow-unauthenticated-identities --cognito-identity-providers ProviderName=cognito-idp.us-east-1.amazonaws.com/awsomestore-pool-id,ClientId=awsomestore-client-id

The aws cognito-identity create-identity-pool command creates a new Amazon Cognito identity pool. To use this command, you need to replace awsomestore-identity-pool with a name of your choice for the identity pool. You also need to replace awsomestore-pool-id with the ID of the Amazon Cognito user pool that you created earlier. Additionally, you need to replace awsomestore-client-id with the ClientId value of the Amazon Cognito user pool client that you created earlier. For example, if you want to create a new Amazon Cognito identity pool called my-identity-pool, and you have an Amazon Cognito user pool with an ID of us-west-2\_abc123xyz and a client ID of 1234567890abcdef, you would use these.

1. Finally, grant permissions to the identity pool:
2. aws cognito-identity set-identity-pool-roles --identity-pool-id awsomestore-identity-pool-id --roles authenticated=arn:aws:iam::<aws\_account\_id>:role/awsomestore-auth-role,unauthenticated=arn:aws:iam::<aws\_account\_id>:role/awsomestore-unauth-role

The aws cognito-identity set-identity-pool-roles command sets the roles for the authenticated and unauthenticated identities in an Amazon Cognito identity pool.

To use this command, you need to replace awsomestore-identity-pool-id with the ID of the Amazon Cognito identity pool that you created earlier. You also need to replace <aws\_account\_id> with your actual AWS account ID and replace awsomestore-auth-role and awsomestore-unauth-role with the names of the IAM roles that you want to use for authenticated and unauthenticated identities, respectively.

For example, if you want to set the roles for an Amazon Cognito identity pool with an ID of us-west-2\_abc123xyz, and you have two IAM roles named my-auth-role and my-unauth-role that you want to use for authenticated and unauthenticated identities, you would replace the relevant fields with these labels..

With the above steps, you can implement authentication and authorization for end customers for the *AWSome Store* website using Amazon Cognito. When using Amazon Cognito for authentication and authorization, it is best to follow these best practices:

* Use MFA to enhance the security of your user pool.
* Implement a password policy requiring strong passwords with letters, numbers, and special characters.
* Regularly monitor the security of your Cognito user pool, such as tracking sign-in attempts and detecting any suspicious activity.
* Use Amazon Cognito’s built-in user sign-up and sign-in process for a smooth and secure experience for your users.
* Use the appropriate user pool attributes to store user data, such as email addresses and phone numbers.
* Use Amazon Cognito’s built-in features for storing user data, such as custom attributes and user pools, to reduce the risk of data breaches.
* Store encrypted sensitive information, such as passwords, in the Amazon Cognito user pool.
* Enable logging and monitoring for Amazon Cognito to help detect security issues and respond to them quickly.
* Consider using Amazon Cognito federated identities for secure **single sign-on** (**SSO**) across multiple AWS services and applications.

Finally, regularly review and update your Amazon Cognito user pool’s security policies to ensure they are up to date with the latest security standards. You can refer to AWS’s user documentation to explore more best practices – <https://docs.aws.amazon.com/cognito/latest/developerguide/multi-tenant-application-best-practices.html>.

Now let’s jump into building your app, starting with the database.

**Define database attributes**

To define database tables and attributes for your *AWSome Store* e-commerce website, you can start by considering the data you need to store in your e-commerce website.

* **Products**: This table can store information about each product available in the store, such as product name, product ID, description, price, image URL, and so on.
* **Customers**: This table can store information about each customer, such as customer name, email address, password, billing address, etc.
* **Orders**: This table can store information about each order placed by a customer, such as order ID, customer ID, product ID, order date, shipping address, and so on.
* **Categories**: This table can store information about product categories, such as category name, category ID, and so on.
* **Inventory**: This table can store information about the inventory of each product, such as product ID, quantity available, and so on.
* **Promotions**: This table can store information about promotions and discounts available for products, such as promotion ID, product ID, discount amount, etc.

When creating these tables, you can set the primary key as the unique identifier for each table, such as product ID, customer ID, and order ID. You can also define secondary indexes to support querying data in your application.

Amazon DynamoDB is a good choice for the *AWSome Store* as you require fast and flexible data storage. DynamoDB is optimized for low latency and high throughput, making it ideal for storing large amounts of data that need to be retrieved quickly. It automatically scales with the growth of your data, allowing you to store and retrieve any amount of data without having to worry about capacity planning. DynamoDB is a cost-effective solution, as you only pay for the read and write capacity that you use, and there are no upfront costs and minimum fees. It integrates seamlessly with other AWS services, making building complex, scalable, and highly available applications easy.

Here is a sample CLI command to create DynamoDB tables and their attributes for the *AWSome Store*:

aws dynamodb create-table \

--table-name awsome\_store\_products \

--attribute-definitions \

AttributeName=product\_id,AttributeType=S \

AttributeName=product\_name,AttributeType=S \

--key-schema \

AttributeName=product\_id,KeyType=HASH \

AttributeName=product\_name,KeyType=RANGE \

--provisioned-throughput \

ReadCapacityUnits=5,WriteCapacityUnits=5

aws dynamodb create-table \

--table-name awsome\_store\_orders \

--attribute-definitions \

AttributeName=order\_id,AttributeType=S \

AttributeName=order\_date,AttributeType=S \

--key-schema \

AttributeName=order\_id,KeyType=HASH \

AttributeName=order\_date,KeyType=RANGE \

--provisioned-throughput \

ReadCapacityUnits=5,WriteCapacityUnits=5

You can create more tables and customize the table names, attributes, and provisioned throughput as per your requirements. The following are some best practices to follow when using DynamoDB:

* **Use partition keys wisely**: Selecting a good partition key that distributes your data evenly is crucial for performance.
* **Consider provisioned throughput**: Ensure you set the right amount of throughput to ensure the required performance.
* **Use secondary indexes**: Secondary indexes can optimize queries based on different attributes.
* **Batch operations**: Use batch operations like BatchGetItem and BatchWriteItem for efficient data retrieval and modification.
* **Use the Time to Live (TTL) attribute**: Use the TTL attribute to delete old or expired items from your table automatically.
* **Store data in a denormalized form**: Storing data in a denormalized form in DynamoDB can simplify and speed up queries.
* **Use automated backups**: Take advantage of DynamoDB’s automatic backups to ensure data durability and reduce the risk of data loss.
* **Monitor and troubleshoot**: Monitor your DynamoDB usage and performance regularly to identify any potential issues and optimize accordingly.
* **Use DynamoDB Streams**: DynamoDB Streams allow you to capture changes made to your DynamoDB tables and process them in real time.

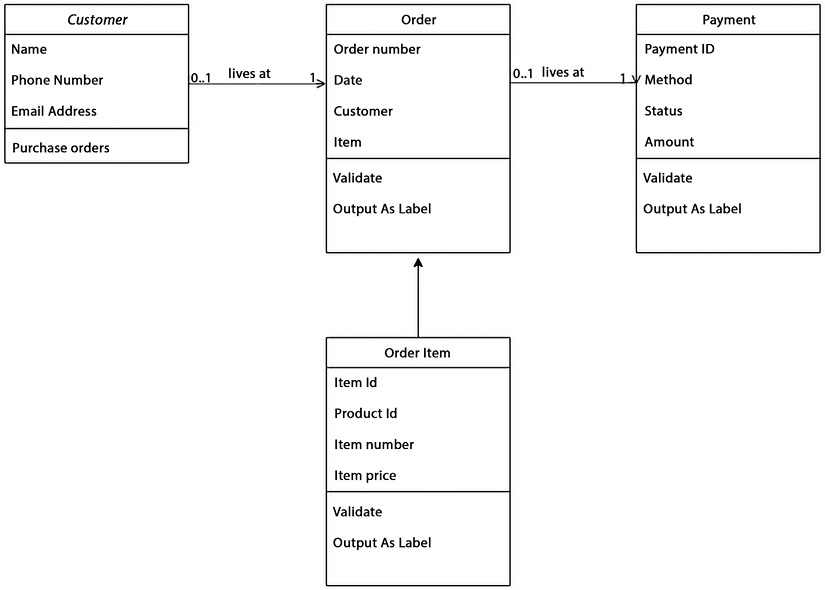
You should use a serverless architecture and AWS Lambda functions to integrate with DynamoDB and offload compute-intensive tasks. You can explore more best practices here – <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-practices.html>. After putting your database together, let’s define Lambda functions to know how to use it.

**Define order context and write AWS Lambda functions**

In the previous section on use cases, we defined the order bounded context in an e-commerce retail application as follows:

* **Order class**: Start by defining the Order class, which represents a customer’s request to purchase one or more products. The Order class could have attributes such as order number, date, customer information, and a list of order items.
* **Order Item class**: Next, define the Order Item class, which represents a single product in an order. The Order Item class could have attributes such as product ID, name, quantity, and price.
* **Customer class**: Also define the Customer class, which represents the person placing the order. The Customer class could have attributes such as customer ID, name, address, and email.
* **Payment class**: Define the Payment class, which represents the payment information for an order. The Payment class could have attributes such as payment method, card number, and expiration date.

Add methods to the classes to represent the actions that can be performed on them. For example, an Order class could have a method for calculating the total cost of the order. Also, an Order class could have a one-to-many relationship with the Order Item class, indicating that an order can have multiple order items as shown below:



*Figure 16.7: Order context class diagram*

Here is an example of how you could implement the Order, OrderItem, Payment, and Customer classes in Node.js using AWS Lambda functions:

const AWS = require('aws-sdk');

class Order {

constructor(orderId, customerId, orderDate, items, payment) {

this.orderId = orderId;

this.customerId = customerId;

this.orderDate = orderDate;

this.items = items;

this.payment = payment;

}

save() {

const dynamoDB = new AWS.DynamoDB.DocumentClient();

const params = {

TableName: 'AwesomeStoreOrders',

Item: {

orderId: this.orderId,

customerId: this.customerId,

orderDate: this.orderDate,

items: this.items,

payment: this.payment

}

};

return dynamoDB.put(params).promise();

}

}

class OrderItem {

constructor(productId, quantity) {

this.productId = productId;

this.quantity = quantity;

}

}

class Payment {

constructor(paymentId, amount, paymentDate, paymentMethod) {

this.paymentId = paymentId;

this.amount = amount;

this.paymentDate = paymentDate;

this.paymentMethod = paymentMethod;

}

}

class Customer {

constructor(customerId, firstName, lastName, email) {

this.customerId = customerId;

this.firstName = firstName;

this.lastName = lastName;

this.email = email;

}

save() {

const dynamoDB = new AWS.DynamoDB.DocumentClient();

const params = {

TableName: 'AwesomeStoreCustomers',

Item: {

customerId: this.customerId,

firstName: this.firstName,

lastName: this.lastName,

email: this.email

}

};

return dynamoDB.put(params).promise();

}

}

This code defines the classes Order, OrderItem, Payment, and Customer with their respective properties. The save method in each class uses the AWS SDK for JavaScript to interact with Amazon DynamoDB and store instances of the class as items in a DynamoDB table. Each save method defines the table’s name in the TableName parameter. You can modify this code to fit your specific needs and add more functionality. Here are some best practices for AWS Lambda:

* **Keep functions small and focused**: Minimize the code in each function and make it perform a single, specific task.
* **Use environment variables for configuration**: Store configuration values as environment variables to avoid hardcoding values in code. Parameter Store is better suited for managing application configuration data that needs to be shared across multiple hosts or environments, requires encryption, and needs versioning and auditing capabilities.

Environment variables, on the other hand, are simpler to use and can be useful for storing smaller amounts of configuration data that only need to be available to a single process or host.

* **Design for statelessness**: Lambda functions should be stateless, meaning they do not persist data between invocations.
* **Optimize cold start time**: Reduce the time it takes for your function to start by minimizing the amount of initialization code and reducing the size of the deployment package.
* **Use VPC for network isolation**: If your function needs to access resources in a VPC, use VPC connectivity to provide network isolation.
* **Use versioning and aliases**: Use versioning and aliases to manage multiple versions of your functions and deploy changes incrementally.
* **Secure secrets and sensitive data**: Store encryption keys and sensitive data in AWS **Key Management Service** (**KMS**) or **AWS Secrets Manager**, rather than in the function code.
* **Monitor function performance**: Use CloudWatch metrics and X-Ray to monitor function performance, and track requests and errors.
* **Automate deployment**: Use AWS CloudFormation, AWS CodeDeploy, or other tools to automate deployment and reduce manual error.

Finally, you should regularly test and validate functions in development and production to validate that they are functioning as intended. You can refer to more Lambda best practices here – <https://docs.aws.amazon.com/lambda/latest/dg/best-practices.html>. Now that you’ve written your code and built your table, let’s deploy and test your code.

**Deploy and test**

Here are the steps to deploy a Lambda function for the *AWSome Store* and connect it with API Gateway:

1. Compress your Lambda function code and any required dependencies into a .zip file.
2. Use the AWS CLI aws lambda create-function command to create a new Lambda function in your AWS account. In this command, you need to specify the name of the function, the runtime environment (Node.js), the path to the .zip file, the name of the handler function, and the IAM role that your function will use to execute. Here is some example code:
3. aws lambda create-function \
4. --function-name my-nodejs-function \
5. --runtime nodejs14.x \
6. --handler index.handler \
7. --memory-size 128 \
8. --zip-file fileb://lambda.zip \
9. --role arn:aws:iam::<aws\_account\_id>:role/my-lambda-role

This command creates a new Lambda function named my-nodejs-function using the Node.js 14.x runtime. The index.handler specifies the entry point to the function code within the lambda.zip file, which is uploaded as the function’s code. The memory-size specifies the amount of memory in megabytes allocated to the function. The role parameter specifies the IAM role that the function uses to access other AWS resources.

1. Use the aws apigateway create-rest-api command to create a new REST API in API Gateway. In this command, you need to specify the name of the API, the description, and the endpoint type (REGIONAL). Here is an example code:
2. aws apigateway create-rest-api --name MyRestApi --description "My new REST API"

In this example, the command creates a new REST API with the name MyRestApi and the description *My new REST API*. You can then use other API Gateway commands to add resources, methods, and integrations to this API.

1. Use the aws apigateway create-resource command to create a new resource in the REST API. In this command, you need to specify the parent resource, the path, and the REST API ID. Here is syntax:
2. aws apigateway create-resource --rest-api-id <rest-api-id> --parent-id <parent-resource-id> --path-part <path-part>

In this command, you need to replace <rest-api-id> with the ID of the REST API that the resource belongs to. You also need to replace <parent-resource-id> with the ID of the parent resource of the new resource you’re creating. The <path-part> parameter specifies the last part of the resource’s path. For example, if you wanted to create a new resource with the path /products, and its parent resource had an ID of abc123, the command would look like this:

aws apigateway create-resource \

--rest-api-id abcd1234 \

--parent-id abc123 \

--path-part products

This would create a new resource with the path /products under the parent resource with ID abc123.

1. Use the aws apigateway put-method command to create a new method (such as GET or POST) for the resource. In this command, you need to specify the REST API ID, the resource ID, and the HTTP method. Here is example code:
2. aws apigateway put-method \
3. --rest-api-id <rest-api-id> \
4. --resource-id <resource-id> \
5. --http-method <http-method> \
6. --authorization-type <authorization-type> \
7. --request-parameters <request-parameters>

In this command, you need to replace <rest-api-id> with the ID of the REST API that the resource belongs to. You also need to replace <resource-id> with the ID of the resource that the method belongs to. The <http-method> parameter specifies the HTTP method (e.g., GET, POST, PUT, DELETE) for the method. The <authorization-type> parameter specifies the type of authorization used for the method (e.g., NONE, AWS\_IAM, CUSTOM). The <request-parameters> parameter is a JSON object that specifies the request parameters accepted by the method.

For example, if you wanted to create a new method for a resource with the ID abc123 that responds to HTTP GET requests and uses AWS IAM authorization, the command would look like this:

aws apigateway put-method \

--rest-api-id abcd1234 \

--resource-id abc123 \

--http-method GET \

--authorization-type AWS\_IAM \

--request-parameters '{"method.request.querystring.foo": true, "method.request.querystring.bar": true}'

This would create a new method for the GET HTTP method that accepts two query string parameters (foo and bar) and uses AWS IAM authorization.

1. Use the aws apigateway put-integration command to integrate the API method with your Lambda function. In this command, you need to specify the REST API ID, the resource ID, the HTTP method, the type of integration, and the ARN of the Lambda function.
2. aws apigateway put-integration \
3. --rest-api-id <rest-api-id> \
4. --resource-id <resource-id> \
5. --http-method <http-method> \
6. --type <type> \
7. --integration-http-method <integration-http-method> \
8. --uri <uri> \
9. --credentials <arn-of-iam-role>

Here’s a brief description of the parameters used:

* + rest-api-id: The ID of the REST API in API Gateway to which you want to add the integration.
  + resource-id: The ID of the resource to which you want to add the integration.
  + http-method: The HTTP method for which you want to create an integration.
  + type: The type of the integration. This can be HTTP for integrating with an HTTP backend or AWS for integrating with an AWS service.
  + integration-http-method: The HTTP method used to call the backend. This is typically POST, GET, PUT, DELETE, or PATCH.
  + uri: The **Uniform Resource Identifier** (**URI**) of the backend service or resource. For example, if you’re integrating with an HTTP backend, this would be the URL of the HTTP endpoint.
  + credentials: The ARN of an IAM role that API Gateway assumes when calling the backend service or resource.

Here is a sample code:

aws apigateway put-integration \

--rest-api-id abcdef1234 \

--resource-id abc123 \

--http-method POST \

--type AWS\_PROXY \

--integration-http-method POST \

--uri arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/arn:aws:lambda:us-east-1:123456789012:function:myLambdaFunction/invocations \

--passthrough-behavior WHEN\_NO\_MATCH \

--content-handling CONVERT\_TO\_TEXT

This command creates an AWS Lambda proxy integration for a POST method on a specific API Gateway resource. The --rest-api-id parameter specifies the ID of the API Gateway REST API, while the --resource-id parameter specifies the ID of the resource to which the method is attached. The --http-method parameter specifies the HTTP method for the method.

The --type parameter specifies the type of integration. In this case, it is set to AWS\_PROXY, which means that the Lambda function is invoked directly from the API Gateway. The --integration-http-method parameter specifies the HTTP method used to invoke the Lambda function. In this case, it is set to POST.

The --uri parameter specifies the ARN of the Lambda function to integrate with.

The --passthrough-behavior parameter specifies the passthrough behavior for unmapped requests. In this case, it is set to WHEN\_NO\_MATCH. The --content-handling parameter specifies how to handle the request payload. In this case, it is set to CONVERT\_TO\_TEXT.

1. Use the aws apigateway create-deployment command to deploy the API to a stage, such as *prod* or *test*. In this command, you need to specify the REST API ID and the stage’s name:
2. aws apigateway create-deployment --rest-api-id <rest-api-id> --stage-name <stage-name>

Replace <rest-api-id> with the ID of the REST API you want to deploy, and <stage-name> with the name of the deployment stage you want to create.

You can also include additional parameters to customize the deployment, such as the description of the deployment, stage variables, and tags. For more information on the available parameters, you can refer to the AWS CLI documentation for this command.

Here’s an example:

aws apigateway create-deployment --rest-api-id abc123 --stage-name prod --description "Production deployment"

This command creates a new deployment of the API with ID abc123 to the prod stage with the description *Production deployment*.

1. Finally, use the CLI aws apigateway get-invoke-url CLI command to get the URL of the deployed API. You can then test the API using this URL. Here is the syntax:
2. aws apigateway get-invoke-url --region <region> --rest-api-id <rest-api-id> --stage-name <stage-name>

Where:

* + <region> is the AWS region where the API is deployed.
  + <rest-api-id> is the ID of the REST API for which to retrieve the endpoint URL.
  + <stage-name> is the name of the API deployment stage for which to retrieve the endpoint URL.

The command returns the public URL of the API endpoint that can be used to make HTTP requests to the API. This URL can be used by clients to access the API.

DevOps is an important step toward automating the entire software delivery pipeline and improving efficiency. Let’s learn about DevOps in detail.

**DevOps in AWS**

DevOps is a methodology that integrates cultural philosophies, practices, and tools to enhance an organization’s capacity to deliver applications and services at a rapid pace. It helps the organization to evolve and improve products faster than traditional software development and infrastructure management processes. This speed allows organizations to serve their customers better and compete in the market more effectively.

To build a **CI/CD** (**Continuous Integration/Continuous Deployment**) pipeline for the *AWSome Store* app, follow these steps:

1. **Code repository**: Store the source code of the *AWSome Store* app in a version control system such as AWS CodeCommit or GitHub/GitLab. Ensure your code is stored in a branch suitable for production deployment.
2. **Build and test**: Use a build tool such as AWS CodeBuild to compile the source code and run tests. This step ensures that the code is stable and working as expected.
3. **Continuous integration**: Integrate the build and test process with the code repository so that the pipeline is triggered automatically when code changes are pushed to the repository. AWS provides several tools to support continuous integration and delivery, including AWS CodePipeline, AWS CodeBuild, and AWS CodeDeploy. These tools enable teams to automate their applications’ building, testing, and deployment.
4. **Monitoring**: Use AWS CloudWatch to monitor the app’s health and detect and alert on any issues. AWS provides various monitoring and logging services, including Amazon CloudWatch, AWS X-Ray, and AWS CloudTrail. These services enable teams to monitor the performance and health of their applications, diagnose issues, and troubleshoot problems.
5. **Continuous deployment**: Automate the deployment of new releases to the production environment whenever the code is integrated and tested successfully.
6. **Roll back**: Plan to roll back to the previous version of the app in case of any issues with the new release.

By implementing a CI/CD pipeline, you can ensure that the *AWSome Store* app is continuously integrated, tested, deployed, and monitored, which helps improve the reliability and speed of software delivery.

After setting up your code repository, the following are the steps to build a CI/CD pipeline using AWS CodePipeline:

1. **Create a build stage**: The build stage will compile your code and create the artifacts that will be deployed. You can use AWS CodeBuild to create the build stage. You will need to define a buildspec.yml file that specifies the commands to build your code. The artifacts should be stored in an S3 bucket.
2. **Create a deployment stage**: The deployment stage will deploy your code to your production environment. You can use AWS CodeDeploy to create the deployment stage. You will need to define an AppSpec file that specifies how the deployment will be done.
3. **Create a pipeline**: Create a new pipeline in AWS CodePipeline and specify the source code repository, build, and deployment stages. You can configure triggers to start the pipeline automatically when code changes are committed to the repository.
4. **Test the pipeline**: Test your pipeline by committing code changes to the repository and verifying that the pipeline is triggered, builds your code, and deploys it to your production environment.
5. **Configure pipeline notifications**: Configure pipeline notifications to receive notifications when pipeline stages succeed or fail. This will help you monitor your pipeline and respond quickly to any issues.
6. **Fine-tune the pipeline**: As you use the pipeline, you may find areas for improvement. You can fine-tune the pipeline by adjusting the pipeline settings or by adding new stages to the pipeline.

As you can see, building a CI/CD pipeline using AWS CodePipeline requires some initial setup and configuration. However, once the pipeline is set up, it can automate the process of building, testing, and deploying your code, which can save you time and help you deliver software more quickly and reliably.

Rollback planning is an essential aspect of any deployment process, as it allows you to revert changes if something goes wrong. Here are some steps you can follow to plan a rollback:

1. **Define the criteria for a rollback**: Identify the conditions that warrant a rollback, such as a critical error that affects the functionality of your application or a significant decrease in performance. Make sure the criteria are well defined and communicated to all stakeholders.
2. **Identify the rollback process**: Determine the steps needed to perform a rollback. This should include taking backups of your data, rolling back to a previous version of your application or code, and ensuring that all changes made during the deployment are appropriately rolled back.
3. **Prepare the rollback plan**: Document the rollback plan, including all the steps you need to take, the timeline for each step, and the roles and responsibilities of everyone involved. Make sure the plan is easily accessible to all stakeholders and that everyone is aware of their responsibilities.
4. **Test the rollback plan**: Before deploying your application, test it to ensure it works as expected. This will help you identify any issues or gaps in the plan before using it in a real-world scenario.
5. **Communicate the rollback plan**: Communicate the rollback plan to all stakeholders, including developers, QA, operations, and management. Ensure everyone knows the plan and what to do in case a rollback is required.
6. **Monitor the deployment**: During the deployment, monitor the performance of your application and keep an eye out for any signs that a rollback may be necessary. This will help you identify issues early and take action before they become critical.

By following these steps, you can ensure that you have a well-defined and tested rollback plan in place, which will help you minimize downtime and avoid potential losses in case something goes wrong during the deployment.

AWS offers a range of adaptable services specifically created to help businesses develop and distribute products more quickly and consistently by utilizing AWS and DevOps methodologies. These services streamline the processes of setting up and maintaining infrastructure, releasing application code, automating software updates, and overseeing the performance of both applications and infrastructure.

**Logging and monitoring**

Setting up logging and monitoring helps organizations better understand their systems, improve their performance and reliability, meet compliance requirements, and enhance the security of their systems. It provides insights into the system’s behavior and performance. This information can be used to identify and troubleshoot issues more quickly. Monitoring the system for potential issues and triggering alerts makes it easier to identify and resolve problems before they become critical.

Logging and monitoring provide detailed performance data that can be used to identify and resolve bottlenecks, leading to improved system performance. They help ensure that the system is meeting compliance requirements and provide evidence in the event of an audit. Monitoring also helps detect and respond to security incidents, protecting sensitive data and maintaining the system’s integrity. Here are the high-level steps to set up logging and monitoring for the *AWSome Store* app in AWS using the CLI:

1. Create a CloudWatch Logs group for your Lambda function.
2. aws logs create-log-group --log-group-name awsomestore-log-group
3. Enable AWS X-Ray for your Lambda function:

To enable AWS X-Ray for a Lambda function, you can use the following command:

aws lambda update-function-configuration --function-name <function-name> --tracing-config Mode=Active

Replace <function-name> with the name of your Lambda function. This command enables active tracing for the Lambda function using AWS X-Ray.

1. Create a CloudWatch alarm to monitor the error rate of your Lambda function:
2. aws cloudwatch put-metric-alarm
3. --alarm-name awsomestore-error-rate # The name of the alarm
4. --comparison-operator GreaterThanThreshold # The comparison operator
5. --evaluation-periods 1 # The number of periods to evaluate the alarm
6. --metric-name Errors # The metric to evaluate
7. --namespace AWS/Lambda # The namespace of the metric
8. --period 300 # The period of the metric in seconds
9. --statistic SampleCount # The statistic to apply to the metric
10. --threshold 1 # The threshold for the alarm
11. --alarm-actions arn:aws:sns:us-east-1:123456789012:awsomestore-alerts # The ARN of the SNS topic to send notifications
12. --dimensions FunctionName=awsomestore # The dimensions to apply to the metric
13. --treat-missing-data breaching # The action to take if data is missing

In this command, you need to replace awsomestore with the name of your Lambda function and 123456789012 with your AWS account ID. You also need to replace arn:aws:sns:us-east-1:123456789012:awsomestore-alerts with the ARN of the SNS topic to which you want to send notifications.

1. View and analyze your logs and metrics in the CloudWatch console.

Configure your Lambda function to send logs and metrics to CloudWatch. You can use the following AWS CLI command:

aws lambda update-function-configuration --function-name <function-name> --handler <handler> --role <role-arn> --environment Variables={LOG\_GROUP\_NAME=/aws/lambda/<function-name>,METRICS\_NAMESPACE=<namespace>}

Here, you need to replace <function-name> with the name of your Lambda function, <handler> with the name of your function’s handler, <role-arn> with the ARN of the execution role for your Lambda function, and <namespace> with the namespace for your CloudWatch metrics.

You can also replace the LOG\_GROUP\_NAME environment variable with the name of the CloudWatch Logs group where you want to send your function’s logs.

To view logs and metrics in the CloudWatch console, here are the steps:

1. Open the AWS Management Console and navigate to the CloudWatch service.
2. In the left sidebar, click on **Logs** to view logs or **Metrics** to view metrics.
3. To view logs, click on the log group associated with your Lambda function, then select a log stream to view the logs.
4. To view metrics, select **Lambda** as the namespace, then select your function name and the metric you want to view (such as **Errors**).
5. You can adjust the time range using the drop-down menu at the top-right corner of the page.

Here are some best practices for logging and monitoring in AWS:

* **Centralized logging**: Centralize logs from multiple services and resources in a single location such as Amazon CloudWatch Logs.
* **Automated alerting**: Set up automated alerts to notify you of potential issues and failures in real time.
* **Real-time monitoring**: Use real-time monitoring tools like Amazon CloudWatch metrics, Amazon CloudWatch alarms, and Amazon CloudWatch dashboards to track key performance indicators.
* **Log retention policy**: Define a log retention policy to ensure that logs are stored for sufficient time to meet compliance and business requirements.
* **Security logging**: Enable security logging for all critical resources to detect and respond to security incidents. You can enable AWS CloudTrail to log API calls made to the resource, enable Amazon S3 server access logging to log requests made to an S3 bucket, and enable VPC Flow Logs to log network traffic to and from an EC2 instance.
* **Monitoring of third-party services**: Monitor the performance and health of third-party services and dependencies to ensure a smooth end-to-end experience.
* **Error logging and debugging**: Ensure detailed error logs are captured and stored for debugging purposes.
* **Log analysis tools**: Use log analysis tools like CloudWatch Log Insights, Amazon Athena, and Amazon QuickSight to analyze log data and identify trends and patterns.
* **Monitoring of resource utilization**: Monitor resource utilization of critical services to ensure they are running optimally and within budget.

You can explore more logging and monitoring best practices here – <https://docs.aws.amazon.com/prescriptive-guidance/latest/logging-monitoring-for-application-owners/logging-best-practices.html>.

This section took you through the high-level steps to build an app. You can extend the provided AWS infrastructure and code based upon your needs. You can refer to the AWS Builder library to learn about implementing a different pattern – <https://aws.amazon.com/builders-library>. Also, there are multiple cloud solutions available to explore from AWS as per your industry use case here – <https://aws.amazon.com/industries/>.

**Optimization with Well-Architected Review**

You learned about the well-architected review in *Chapter 2*, *Understanding the AWS Well-Architected Framework and Getting Certified*. A well-architected review report is a comprehensive review of your AWS infrastructure and applications, designed to help you improve the robustness, security, and performance of your solutions.

A typical well-architected review report would include a summary of your current architecture and recommendations for improvement in each of the six pillars. The report would also include a list of best practices and guidelines for ensuring that your solutions are well-architected. Here are some best practices to consider when setting up the AWS infrastructure for your use case:

* Use AWS Organizations to manage multiple AWS accounts for production, testing, and development purposes. In this chapter, you deployed your app in a single dev account, but for production and test environments, you may want to have separate accounts and manage them using AWS Organizations.
* Use IAM policies and roles to control access to AWS resources and limit the permissions of users and services.
* Use **Virtual Private Clouds** (**VPCs**) to create a virtual network with a logically isolated section of the AWS cloud where you can launch AWS resources. You learned about AWS networking in *Chapter 4*, *Networking in AWS*. In this chapter, to keep things simple, we have not covered networking, but you can build VPCs and put Lambda inside them for security, especially for customer management functions.
* Use Amazon S3 to store user data, application backups, and website hosting.
* Use Amazon DynamoDB for NoSQL data storage to store information such as customer profiles, orders, and products.
* Use Amazon API Gateway to create, deploy, and manage APIs for your application.
* Use Amazon Lambda to run your application code and to execute business logic.
* Use Amazon CloudWatch to monitor, troubleshoot, and alert on the performance of your AWS resources.
* Use Amazon CloudFront for content delivery and to distribute your application to multiple locations for low latency and high performance.
* Use AWS Certificate Manager to secure your website using SSL/TLS certificates.
* Use Amazon Route 53 for domain name registration and for routing users to your application.
* Implement disaster recovery strategies, such as backups and multiple availability zones.
* Monitor and manage costs with the AWS Cost Explorer and monitor security with AWS Security Hub.

Finally, adhere to the AWS Well-Architected Framework and perform regular reviews to ensure your infrastructure is secure, reliable, and cost-effective.

**Summary**

In this chapter, you have put together many of the technologies, best practices, and AWS services we have covered in this book. You weaved it together into an e-commerce website architecture that you should be able to leverage and use for your own future projects.

You built architecture using AWS services following domain driven-design. You learned about implementing various AWS services, including IAM, S3, DynamoDB, Lambda, and API Gateway, and using the AWS CLI. You learned several best practices, along with a well-architected framework to optimize your architecture.

As fully featured as AWS has become, it will continue providing more services to help large and small enterprises simplify their information technology infrastructure. You can rest assured that AWS is creating new services and improving the existing services by making them better, faster, easier, more flexible, and more powerful, as well as by adding more features.

As of 2023, AWS offers 200+ services. That’s a big jump from the two services it offered in 2004. AWS’s progress in the last 18 years has been monumental. I cannot wait to see what the next 18 years will bring for AWS and what kind of solutions we can deliver with their new offerings.

I hope you are as excited as I am about the possibilities that these new services will bring.