

Building well-architected serverless applications: Approaching application lifecycle management – part 1

by Julian Wood | on 15 JUN 2020 | in [Amazon Cognito](#), [AWS Amplify](#), [AWS AppSync](#), [AWS Cloud Development Kit](#), [AWS CloudFormation](#), [AWS Lambda](#), [AWS Serverless Application Model](#), [AWS Well-Architected Tool](#), [Serverless](#) | [Permalink](#) | [Share](#)

This series of blog posts uses the [AWS Well-Architected Tool](#) with the [Serverless Lens](#) to help customers build and operate applications using best practices. In each post, I address the nine serverless-specific questions identified by the Serverless Lens along with the recommended best practices. See the [Introduction post](#) for a table of contents and explanation of the example application.

Question OPS2: How do you approach application lifecycle management?

Adopt lifecycle management approaches that improve the flow of changes to production with higher fidelity, fast feedback on quality, and quick bug fixing. These practices help you rapidly identify, remediate, and limit changes that impact customer experience. By having an approach to application lifecycle management, you can reduce errors caused by manual process and increase the levels of control to gain confidence your workload operates as intended.

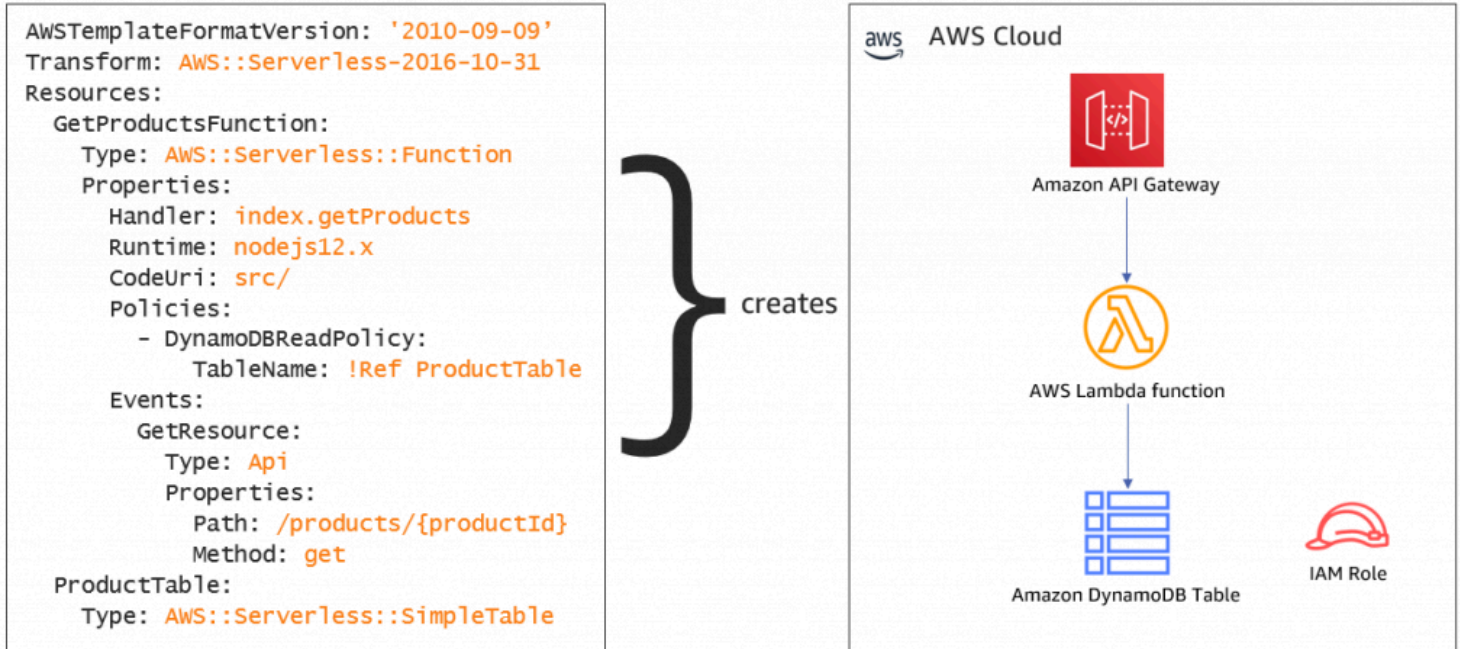
Required practice: Use infrastructure as code and stages isolated in separate environments

Infrastructure as code is a process of provisioning and managing cloud resources by storing application configuration in a template file. Using infrastructure as code helps to deploy applications in a repeatable manner, reducing errors caused by manual processes such as creating resources in the AWS Management Console.

Storing code in a version control system enables tracking and auditing of changes and releases over time. This is used to roll back changes safely to a known working state if there is an issue with an application deployment.

Infrastructure as code

For AWS Cloud development the built-in choice for infrastructure as code is [AWS CloudFormation](#). The template file, written in JSON or YAML, contains a description of the resources an application needs. CloudFormation automates the deployment and ongoing updates of the resources by creating CloudFormation stacks.



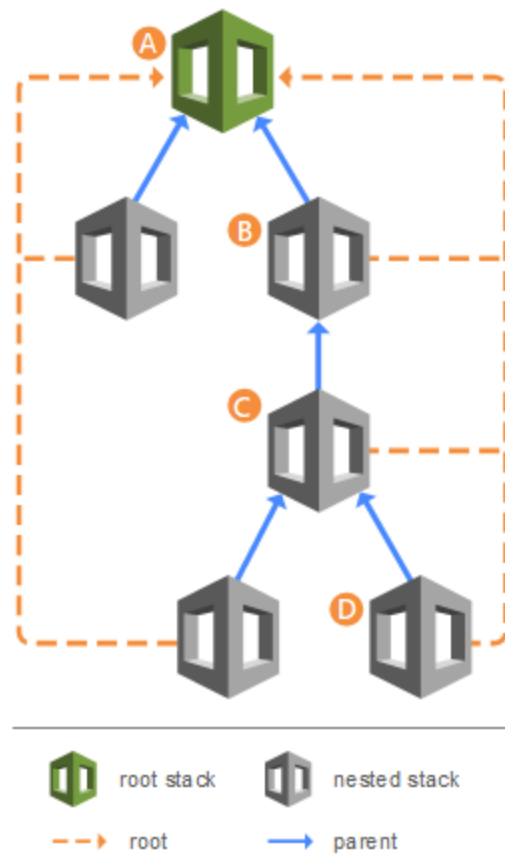
CloudFormation code example creating infrastructure

There are a number of higher-level tools and frameworks that abstract and then generate CloudFormation. A serverless specific framework helps model the infrastructure necessary for serverless workloads, providing either declarative or imperative mechanisms to define event sources for functions. It wires permissions between resources automatically, adds resource configuration, code packaging, and any infrastructure necessary for a serverless application to run.

The [AWS Serverless Application Model](#) (AWS SAM) is an AWS open-source framework optimized for serverless applications. The [AWS Cloud Development Kit](#) allows you to provision cloud resources using familiar programming languages such as TypeScript, JavaScript, Python, Java, and C#/.Net. There are also third-party solutions for creating serverless cloud resources such as the [Serverless Framework](#).

The [AWS Amplify Console](#) provides a git-based workflow for building, deploying, and hosting serverless applications including both the frontend and backend. The [AWS Amplify CLI](#) toolchain enables you to add backend resources using CloudFormation.

For a large number of resources, consider breaking common functionality such as monitoring, alarms, or dashboards into separate infrastructure as code templates. With CloudFormation, use [nested stacks](#) to help deploy them as part of your serverless application stack. When using AWS SAM, import these nested stacks as [nested applications](#) from the AWS Serverless Application Repository.



AWS CloudFormation nested stacks

Here is an example [AWS SAM template using nested stacks](#). There are two `AWS::Serverless::Application` nested resources, [api.template.yaml](#) and [database.template.yaml](#). For more information on nested stacks, see the [AWS Partner Network](#) blog post: [CloudFormation Nested Stacks Primer](#).

Version control

The [serverless airline example](#) application used in this series uses Amplify Console to provide part of the backend resources, including authentication using [Amazon Cognito](#), and a GraphQL API using [AWS AppSync](#).

The [airline application code](#) is stored in GitHub as a version control system. Fork, or copy, the application to your GitHub account. Configure Amplify Console to connect to the GitHub fork.

When pushing code changes to a fork, Amplify Console automatically deploys these backend resources along with the rest of the application. It hosts the application at the *Production branch URL*, and you can also configure a custom domain name if needed.

App details

App name

awsserverlessairline

Source repository

<https://github.com/julianwood/aws-serverless-airline-booking/tree/master> 

Production branch URL

<https://develop.██████████.amplifyapp.com> 

Framework

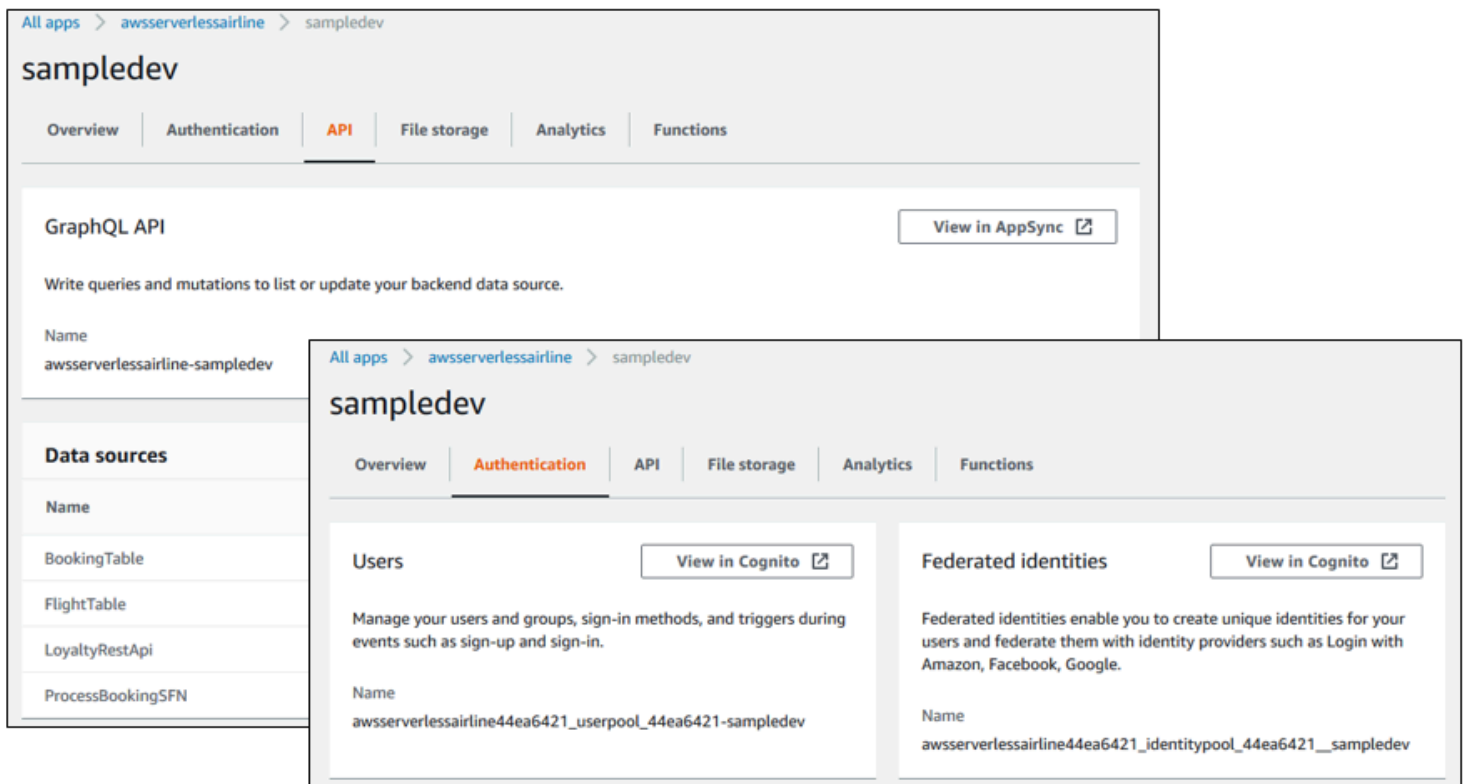
Web - Amplify

AWS Amplify Console App details

The Amplify Console configuration to create the API and Authentication backend resources is found in the [backend-config.json](#) file. The resources are provisioned during the Amplify Console [build](#) phase.

To view the deployed resources, within the [Amplify Console](#), navigate to the *awsserverlessairline* application. Select **Backend environments** and then select an environment, in this example *sampledev*.

Select the **API** and **Authentication** tabs to view the created backend resources.



The screenshot displays the AWS Amplify Console interface for the *sampledev* environment. The top navigation bar shows the breadcrumb *All apps > awsserverlessairline > sampledev*. The main header is **sampledev**. Below the header, there are tabs for **Overview**, **Authentication**, **API**, **File storage**, **Analytics**, and **Functions**. The **API** tab is currently selected, showing the **GraphQL API** section. It includes a button **View in AppSync** and a description: "Write queries and mutations to list or update your backend data source." Below this, the **Data sources** table is visible, listing *BookingTable*, *FlightTable*, *LoyaltyRestApi*, and *ProcessBookingSFN*. The **Authentication** tab is also shown, displaying the **Users** and **Federated identities** sections. The **Users** section includes a button **View in Cognito** and a description: "Manage your users and groups, sign-in methods, and triggers during events such as sign-up and sign-in." Below this, the **Name** field shows *awsserverlessairline44ea6421_userpool_44ea6421-sampledev*. The **Federated identities** section includes a button **View in Cognito** and a description: "Federated identities enable you to create unique identities for your users and federate them with identity providers such as Login with Amazon, Facebook, Google." Below this, the **Name** field shows *awsserverlessairline44ea6421_identitypool_44ea6421__sampledev*.

AWS Amplify Console deployed backend resources

Using multiple tools

Applications can use multiple tools and frameworks even within a single project to manage the infrastructure as code. Within the airline application, [AWS SAM](#) is also used to provision the rest of the serverless infrastructure using nested stacks. During the Amplify Console [build process](#), the [Makefile](#) contains the AWS SAM build instructions for each application service.

For example, the AWS SAM build instructions to deploy the booking service are as follows:

```
Bash
deploy.booking: ##=> Deploy booking service using SAM
$(info [*] Packaging and deploying Booking service...)
cd src/backend/booking && \
    sam build && \
    sam package \
        --s3-bucket ${DEPLOYMENT_BUCKET_NAME} \
        --output-template-file packaged.yaml && \
    sam deploy \
        --template-file packaged.yaml \
        --stack-name ${STACK_NAME}-booking-${AWS_BRANCH} \
        --capabilities CAPABILITY_IAM \
        --parameter-overrides \
            BookingTable=${AWS_BRANCH}/service/amplify/storage/table/booking \
            FlightTable=${AWS_BRANCH}/service/amplify/storage/table/flight \
            CollectPaymentFunction=${AWS_BRANCH}/service/payment/function/collect \
            RefundPaymentFunction=${AWS_BRANCH}/service/payment/function/refund \
            AppsyncApiId=${AWS_BRANCH}/service/amplify/api/id \
            Stage=${AWS_BRANCH}
```

Each service has its own AWS SAM template.yml file. The files contain the resources for each of the [booking](#), [catalog](#), [log-processing](#), [loyalty](#), and [payment](#) services. This means that the services can be managed independently within the application as separate stacks. In larger applications, these services may be managed by separate teams, or be in separate repositories, environments or AWS accounts. It may make sense to split out some common functionality such as alarms, or dashboards into separate infrastructure as code templates.

AWS SAM can also use IAM roles to assume temporary credentials and deploy a serverless application to separate AWS accounts.

For more information on managing serverless code, see [Best practices for organizing larger serverless applications](#).

View the deployed resources in the [AWS CloudFormation Console](#). Select **Stacks** from the left-side navigation bar, and select the **View nested** toggle.

	Stack name	Status
<input type="radio"/>	amplify-awsserverlessairline-sampledev-131601-log-proc-LambdaInvocationCustomResource-B0OLG7Q99T0B NESTED	✔ UPDATE_COMPLETE
<input type="radio"/>	amplify-awsserverlessairline-sampledev-131601-log-proc-SubscribeLogsToProcessingStream-1GEGXD96UDGM NESTED	✔ UPDATE_COMPLETE
<input type="radio"/>	amplify-awsserverlessairline-sampledev-131601-log-processing-ProcessCustomMetricsAsync-3DIPY7Y882E8 NESTED	✔ UPDATE_COMPLETE
<input type="radio"/>	amplify-awsserverlessairline-sampledev-131601-log-proce-LambdaInvocationCustomResource-16VJJ20JBRXFR NESTED	✔ UPDATE_COMPLETE
<input type="radio"/>	amplify-awsserverlessairline-sampledev-131601-log-processing-AutoSetLogGroupsRetention-1DJD8G2CUX066 NESTED	✔ UPDATE_COMPLETE
<input type="radio"/>	amplify-awsserverlessairline-sampledev-131601-log-processing-develop	✔ UPDATE_COMPLETE
<input type="radio"/>	amplify-awsserverlessairline-sampledev-131601-loyalty-develop	✔ UPDATE_COMPLETE
<input type="radio"/>	amplify-awsserverlessairline-sampledev-131601-booking-develop	✔ UPDATE_COMPLETE
<input type="radio"/>	amplify-awsserverlessairline-sampledev-131601-payment-develop-StripePaymentApplication-1OCEEG68UITW1 NESTED	✔ UPDATE_COMPLETE
<input type="radio"/>	amplify-awsserverlessairline-sampledev-131601-payment-develop	✔ UPDATE_COMPLETE
<input type="radio"/>	amplify-awsserverlessairline-sampledev-131601-apiawsserverlessairl-CustomResourcesjson-T4MIC6IC28KZ NESTED	✔ CREATE_COMPLETE
<input type="radio"/>	amplify-awsserverlessairline-sampledev-131601-apiawsserverlessairline-ConnectionStack-I8MSW064859V NESTED	✔ CREATE_COMPLETE
<input type="radio"/>	amplify-awsserverlessairline-sampledev-131601-apiawsserverlessairline-1X5FEIA64-Flight-1LU0O6BUUWZTM NESTED	✔ CREATE_COMPLETE
<input type="radio"/>	amplify-awsserverlessairline-sampledev-131601-apiawsserverlessairline-1X5FEIA6-Booking-PKJU6QATFZRV NESTED	✔ CREATE_COMPLETE
<input type="radio"/>	amplify-awsserverlessairline-sampledev-131601-apiawsserverlessairline-1X5FEIA64YSK2 NESTED	✔ CREATE_COMPLETE
<input type="radio"/>	amplify-awsserverlessairline-sampledev-131601-authawsserverlessairline44ea6421-1XF0YKR33BAF0 NESTED	✔ CREATE_COMPLETE
<input type="radio"/>	amplify-awsserverlessairline-sampledev-131601	✔ UPDATE_COMPLETE

Viewing CloudFormation nested stacks

The serverless airline application is a more complex example application comprising multiple services composed of multiple CloudFormation stacks. Some stacks are managed via Amplify Console and others via AWS SAM. Using infrastructure as code is not only for large and complex applications. As a best practice, we suggest using SAM or another framework for even simple, small serverless applications with a single stack. For a getting started tutorial, see the example [Deploying a Hello World Application](#).

Improvement plan summary

1. Use a serverless framework to help you execute functions locally, build and package application code. Separate packaging from deployment, deploy to isolated stages in separate environments, and support secrets via configuration management systems.
2. For a large number of resources, consider breaking common functionalities such as alarms into separate infrastructure as code templates.

Conclusion

Introducing application lifecycle management improves the development, deployment, and management of serverless applications. In this post I cover using infrastructure as code with version control to deploy applications in a repeatable manner. This reduces errors caused by manual processes and gives you more confidence your application works as expected.

This well-architected question continues in [part 2](#) where I look further at deploying to multiple stages using temporary environments, and rollout deployments.

TAGS: [Amazon Cognito](#), [AppSync](#), [AWS Amplify Console](#), [AWS Cloud Development Kit](#), [AWS CloudFormation](#), [AWS Lambda](#), [AWS Serverless Application Model](#), [serverless](#), [well-architected](#)