aws

Developer Guide

Serverless

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Serverless: Developer Guide

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

What is serverless development? 1
Serverless data processing 4
Asynchronous processing 4
Synchronous processing 5
Streaming 6
Stateless data 7
Prerequisites 8
Amazon Web Services account 9
Programming languages 12
Development environment 12
AWS cloud infrastructure 13
Regions 13
Amazon Resource Name (ARN) 14
Security model 15
Understanding serverless development 17
Traditional development 17
Use services instead of custom code 19
Serverless development on AWS 20
Transitioning to event-driven architecture 22
Decoupled event-driven architecture 27
Summary 28
Next steps 29
Focusing on core services 30
Common serverless services 30
Networking & content delivery 32
Front-end web & mobile 32
Application integration 32
Database & storage 32
Compute 33
Security, identity & compliance 33
Management & governance 33
Developer tools and code instrumentation 34
Streaming & batch processing 34
Typical microservice example 34

Service starters 36
IAM 36
What is Identity and Access Management? 36
Fundamentals 37
Advanced topics 45
Additional resources 46
Next Steps 46
Lambda 46
What is Lambda? 47
Fundamentals 48
Advanced Topics 57
Additional resources 60
Next steps 60
API Gateway 61
What is API Gateway? 62
Fundamentals 62
Advanced Topics 67
Additional resources 71
Next Steps 71
DynamoDB 72
What is DynamoDB? 72
Fundamentals 74
Advanced Topics 79
Additional resources 83
Next steps 84
Learn using a workshop 85
Document history 87

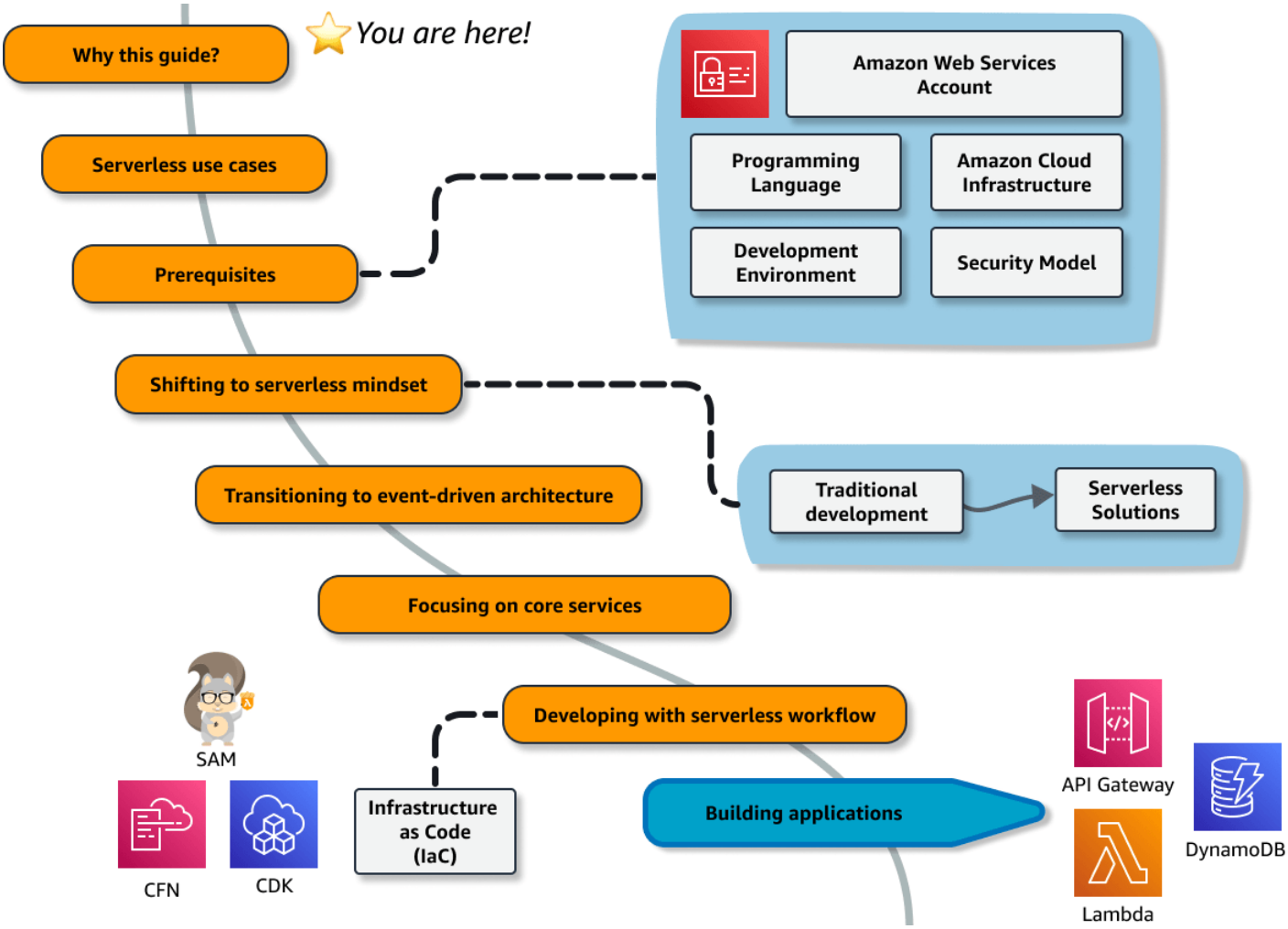
What is serverless development?

The following topics will guide you through developing a better conceptual understanding of serverless application development, and how various AWS services fit into together to create *application patterns* that form the core of your cloud applications. These applications can range from microservices that handle discreet business logic as a part of your application back-end, to event-driven workflows that perform data transformations or processing.

Understanding serverless development will you help you make critical decisions about which AWS services are best suited for your business need. For example, choosing between Amazon DocumentDB, DynamoDB, and Aurora PostgreSQL for a database depends on various factors, such as what type of data-structure you want to use, or how many concurrent database connections you anticipate as your applications scale.

The goal of this serverless developer guide is to give you directed **learning paths** for the core services you need to implement serverless solutions.

Serverless development lets you build applications without managing long-running servers, such as a provisioned Amazon EC2 instance. AWS serverless technologies are pay-as-you-go, can scale up and down as your application needs change, and are built to expand across AWS Regions to ensure resiliency.



This guide will highlight what you need to know right away and link to service documentation for more service-specific details.

For example, you will learn that the Lambda service creates an *execution environment* to run compute functions. For more information on how Lambda manages function scaling or reduces start-up time, we will link you to relevant sections of the Lambda developer guide.

The topics in this guide will cover the prerequisites for understanding serverless development on AWS, such as account creation and an overview of AWS cloud infrastructure. Then, you will learn how to shift from a traditional development model to a serverless, event-driven architecture with which to develop applications on the cloud.

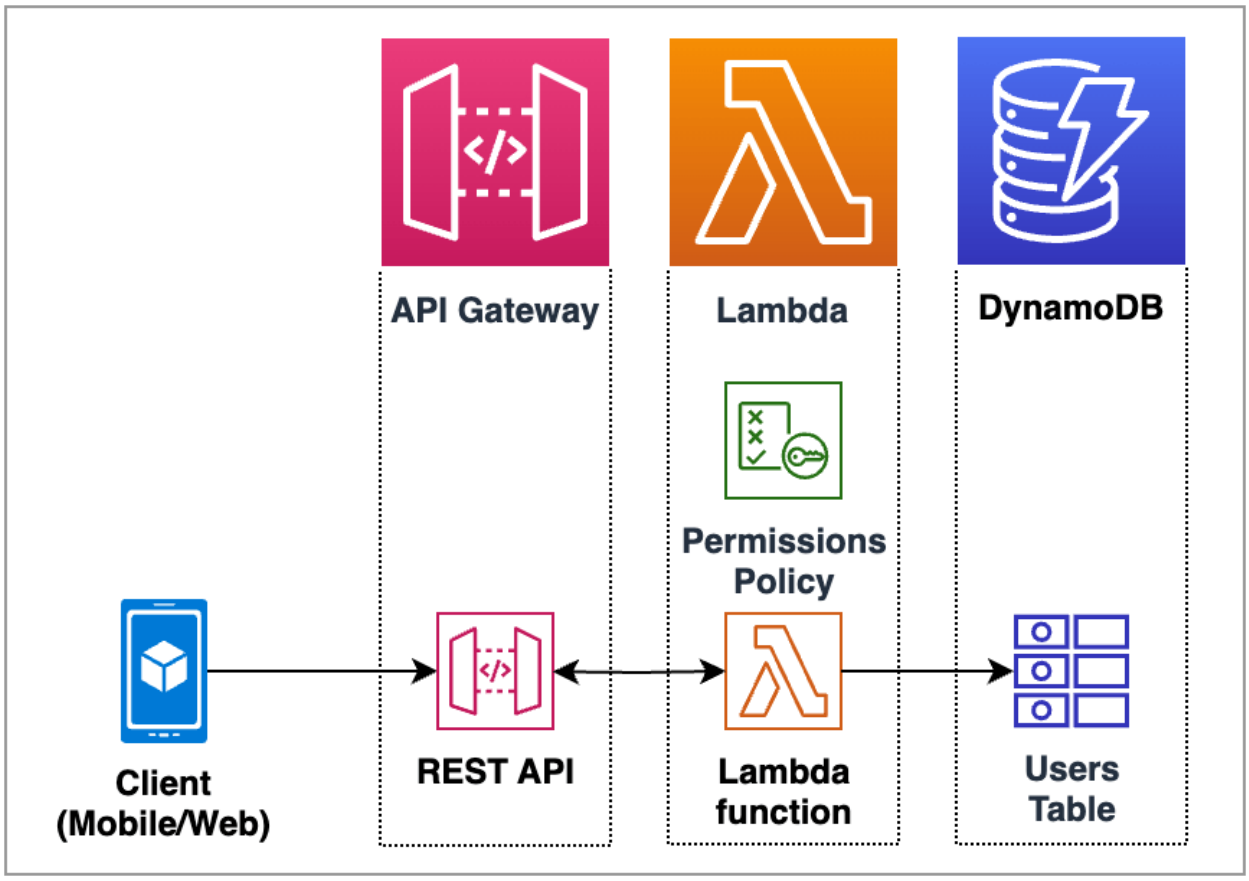
Along the way, this guide will introduce core services, workshops, and tutorials, you can choose to reinforce your learning with hands-on activities.

- AWS Identity and Access Management — for securely accessing resources on AWS.

- AWS Lambda — for serverless compute functionality.
- Amazon API Gateway for integrating HTTP and HTTPS requests with services to handle the requests.
- Amazon DynamoDB for data storage and retrieval

Learn serverless techniques in an online workshop

Learn by doing in the Serverless Patterns Workshop. The first module introduces a serverless microservice to retrieve data from DynamoDB with Lambda and API Gateway. Additional modules provide practical examples of unit and integration testing, using infrastructure as code to deploy resources, and how to build common architectural patterns used in serverless solutions.



Understanding serverless data processing

Processing data in serverless applications largely falls within the following three patterns:

- **Asynchronous processing**– big data processing, image/video manipulation, web hooks
- **Synchronous processing** – web apps, web services, microservices, web hooks
- **Streaming** – processing inbound data streams, from apps, IoT devices

The following topics provide a broad overview of each serverless processing pattern and explain the most common services you can use for each type. Use these topics to gain a conceptual understanding of serverless data processing on AWS.

Topics

- Asynchronous processing
- Synchronous processing
- Streaming
- Stateless data

Asynchronous processing

Serverless development allows your applications to ingest, process and analyze high volumes of data quickly and efficiently.

As the volume of data coming from increasingly diverse sources grows, you might find you need to move quickly to process this data to ensure that your application's business logic can meet your needs. To process data at scale, organizations need to elastically provision resources to manage the information they receive from various microservices, mobile devices, operational data stores, and other sources.

Learn how to build a scalable serverless data processing solution. Use Amazon Simple Storage Service to trigger data processing or load machine learning (ML) models so that Lambda can perform ML inference in real time.

- **File processing** – Suppose you have a photo sharing application. People use your application to upload photos, and the application stores these user photos in an Amazon S3 bucket. Then, your application creates a thumbnail version of each user's photos and displays them on the

user's profile page. In this scenario, you may choose to create a Lambda function that creates a thumbnail automatically. Amazon S3 is one of the supported AWS event sources that can publish *object-created events* and invoke your Lambda function. Your Lambda function code can read the photo object from the Amazon S3 bucket, create a thumbnail version, and then save it in another Amazon S3 bucket.

- **Image identification** – Given the same photo sharing application, suppose now that you want to provide automatic categorization of images for your users. In this scenario, Amazon Rekognition will queue each images for processing. After analysis, faces are detected and your application can implement similarity scores to group photos by family members, for example. Objects, scenes, activities, landmarks, and dominant colors are detected and labels are applied to improve categorization and search.

To implement asynchronous processing in similar scenarios, you can use the following AWS services together.

- **AWS Lambda** — For compute processing tasks.
- **AWS Step Functions** — For managing and orchestrating microservice workflows.
- **Amazon Simple Notification Service** — For message delivery from publishers to subscribers, plus *fan out* which is when a message published to a topic is replicated and pushed to multiple endpoints for parallel asynchronous processing.
- **Amazon Simple Queue Service** — For creating secure, durable, and scalable queues for asynchronous processing.
- **Amazon DynamoDB** and **Amazon S3** — For storing and retrieving data and files

Synchronous processing

Microservice architecture breaks applications into loosely coupled services. Each microservice is independent, making it easy to scale up a single service or function without needing to scale the entire application. Individual services are loosely coupled, letting independent teams focus on a single business process, without the need for them to understand the entire application.

Microservices also let you choose which individual components suit your business needs, giving you the flexibility to change your selection without rewriting your entire workflow. Different teams can use the programming languages and frameworks of their choice to work with their microservice, and this microservice can still communicate with any other in the application through application programming interfaces (APIs).

Examples:

- **Websites** — Suppose you are creating a website and you want to host the back-end logic on Lambda. You can invoke your Lambda function over HTTP using Amazon API Gateway as the HTTP endpoint. Now, your web client can invoke the API, and then API Gateway can route the request to Lambda. You can also implement route authentication and authorization by integrating Amazon Cognito with API Gateway
- **Mobile applications** — Suppose you have a custom mobile application that produces events. You can create a Lambda function to process events published by your custom application. For example, you can configure a Lambda function to process the clicks within your custom mobile application.

To implement synchronous processing in similar scenarios, you can use the following AWS services together.

- **AWS Lambda** — For compute processing tasks.
- **Amazon API Gateway** — For connecting and scaling inbound requests.
- **AWS Step Functions** — For managing and orchestrating microservice workflows.
- **Amazon DynamoDB** & **S3** — For storing and retrieving data and files.
- **Amazon Cognito** for authentication and authorization of users.

Streaming

Streaming data lets you to gather analytical insights from your application and process them in real-time. Streaming typically presents a unique set of design and architectural challenges.

Lambda and Amazon Kinesis can process real-time streaming data for application activity tracking, transaction order processing, click-stream analysis, data cleansing, log filtering, indexing, social media analysis, Internet of Things (IoT) device data telemetry, and metering.

- **Data and analytics** — Suppose you are building an analytics application and storing raw data in a DynamoDB table. When you write, update, or delete items in a table, DynamoDB streams can publish item update events to a stream associated with the table. In this case, the event data provides the item key, event name (such as insert, update, and delete), and other relevant details. You can write a Lambda function to generate custom metrics by aggregating raw data.
- **Monitoring metrics** — Amazon Prime Video monitors metrics from devices worldwide to ensure quality-of-service. The team chose Amazon Kinesis Data Streams to deliver video stream

metadata and to collect metrics. Data is sent to Amazon OpenSearch Service for application monitoring and forensic analysis. The services aggregate, analyze, and visualize data to provide real-time insights that help the team find and fix streaming issues as they happen. For more information on this specific use-case, see Using AWS to Deliver Streaming Experience to More Than 18 Million Football Fans

To implement serverless streaming in similar scenarios, you can use the following AWS services together.

- **AWS Lambda** — For compute processing tasks.
- **Amazon Kinesis** — For collecting, processing, and analyzing real-time and streaming data.
- **Amazon DynamoDB** & **Amazon S3** — For storing and retrieving data and files.

Stateless data

When building Lambda functions, you should assume that the environment exists only for a single invocation. The function should initialize any required state when it is first started – for example, fetching a shopping cart from a DynamoDB table. It should commit any permanent data changes before exiting to a durable store such as Amazon S3, DynamoDB, or Amazon SQS. It should not rely on any existing data structures or temporary files, or any internal state that would be managed by multiple invocations (such as counters or other calculated, aggregate values).

Lambda provides an initializer before the handler where you can initialize database connections, libraries, and other resources. Since execution environments are reused where possible to improve performance, you can amortize the time taken to initialize these resources over multiple invocations. However, you should not store any variables or data used in the function within this global scope.

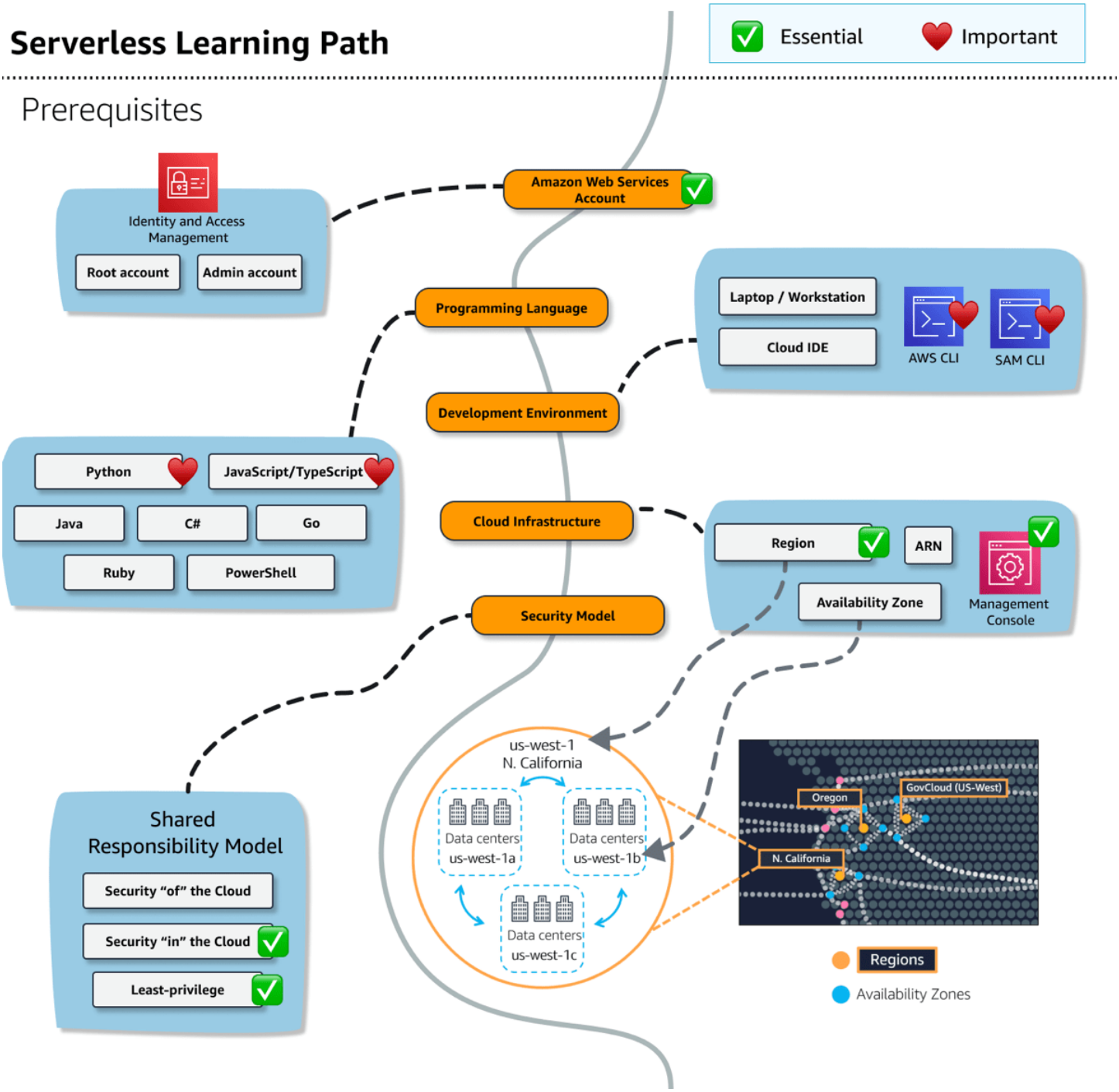
Understanding prerequisites for serverless development

Before you begin to develop a serverless application, there are some key concepts you need to understand. Review the serverless learning path in the following diagram.

Topics are shown in orange boxes. Large topics may be broken down into several sub-topics in blue. Icons represent related services or tools. Essential topics are noted with a green check box. Important, but not essential, items are noted with a red mark. When a high level orange topic is marked as essential, that means all of the sub-topics are essential too.

Topics

- Amazon Web Services account
- Programming languages
- Development environment
- AWS cloud infrastructure
- Security model



Amazon Web Services account

Before getting started, you must have or create an Amazon Web Services (AWS) account.

If you are creating a new account, you will create a root account using an email address. The **root** account has *unrestricted access*, similar to root accounts for an operating system. As a best practice, you should create an administrative user too.

> ⚠ **Granting administrative access to a user**
>
> As you might guess, granting administrative access to a user is still rather far reaching. An account with administrative level privileges will make getting started easier. For systems in production, follow the principle of least-privilege — granting only the minimum access necessary to accomplish tasks.

- For a step-by-step guide to account types and login management, see Signing in to the AWS Management Console.
- AWS Identity and Access Management (IAM) is the service to manage entities and resources authorized to use services and service resources.

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.
2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

Programming languages

We assume that you have some experience with coding and deploying programs using one of the supported languages. This guide will *not* teach you how to program, but it will at times provide code samples.

Writing functions in an interpreted language like Python or JavaScript might be more straight-forward in some scenarios because your code can be added directly through the AWS Management Console web interface.

You can use one of the listed languages, or create your own Lambda runtime container.

- Python, JavaScript/TypeScript — Commonly used interpreted languages
- Java, C#, Go — Compiled languages
- Ruby, PowerShell — Less frequently used options

Development environment

For serverless development, you will likely want to set up and use a familiar editor or IDE, such as Visual Studio Code or PyCharm. Alternatively, you may prefer AWS Cloud9, a browser-based IDE and terminal "in the cloud" with direct access to your AWS account.

You should definitely install the AWS CLI for command line control and automation of services. For example, you can list your Amazon S3 buckets, update Lambda functions, or send test events to invoke service resources. Many tutorials will show how to complete tasks with the AWS CLI.

Another useful, but optional, tool is the AWS Serverless Application Model CLI, aka the "SAM CLI". AWS SAM templates define infrastructure services and code. You can use AWS SAM CLI to build and

deploy from these templates to the cloud. AWS SAM CLI also provides features to test and debug locally and deploy changes to infrastructure and code.

> **Tip**
>
> There are other tools that our customers use, such as the AWS Cloud Development Kit (AWS CDK) for programmatic creation of infrastructure and code, or several 3rd party IaC options.

You will find that some services provide emulators that can run on your local laptop. These tools can be useful for local development, but they are also limited in terms of service and API coverage. Serverless services are better suited to their native cloud environment.

Related resources:

- Install AWS CLI - to control and manage your AWS services from the command line
- Install AWS SAM CLI - to create, deploy, test, and update your serverless code and resources from the command line
- Note: These tools are provided by AWS Cloud9, but you should update to the latest available versions.

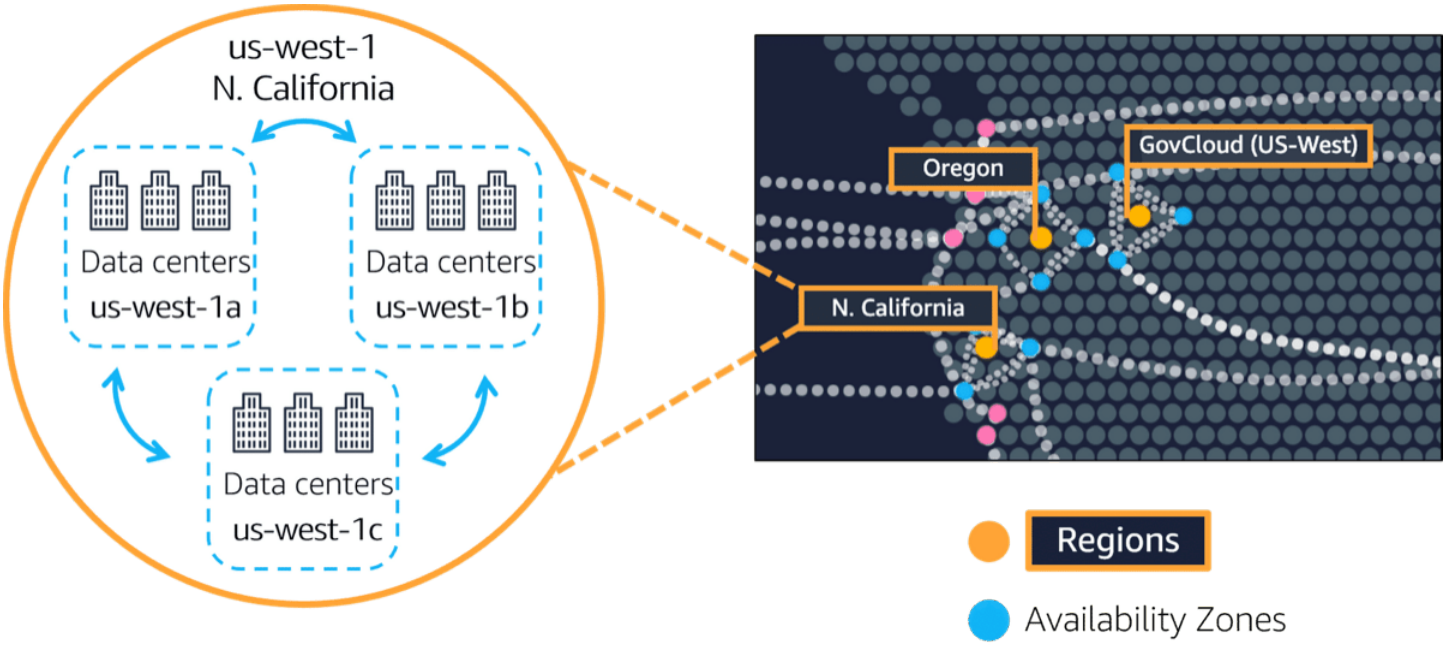
AWS cloud infrastructure

AWS provides services across the globe. You only need to understand how regions, availability zones, and data centers are related so that you can select a region. You will see the region code in URLs and Amazon Resource Names (ARNs), unique identifiers for AWS resources.

Regions

Every solution you build that runs in the AWS cloud will be deployed to at least one region.

- **Region** – a physical location around the world where we cluster data centers
- **Availability Zone** or "AZ" - one or more discrete data centers with redundant power, networking, and connectivity *within* a Region
- **Data center** – a physical location that contains servers, data storage drives, and network equipment



Amazon has many regions all across the globe. Inside each region, there are one or more Availability Zones located tens of miles apart. The distance is near enough for low latency — the gap between requesting and receiving a response, and far enough to reduce the chance that multiple zones are affected if a disaster happens.

Each region is identified by a code, such as "us-west-1", "us-east-1" or "eu-west-2". Within each region, the multiple isolated locations known as *Availability Zones* or AZs are identified with the region code followed by a letter identifier. For example, `us-east-1a`. AWS handles deploying to multiple availability zones within a region for resilience.

Amazon Resource Name (ARN)

Services are identified with regional endpoints. The general syntax of a regional endpoint is as follows:

```
protocol://<service-code>.<region-code>.amazonaws.com
```

For example, `https://dynamodb.us-west-1.amazonaws.com` is the endpoint for the Amazon DynamoDB service in the US West (N. California) Region.

The region code is also used to identify AWS resources with Amazon Resource Names, also called "ARNs". Because AWS is deployed all over the world, ARNs function like an addressing system to precisely locate which specific part of AWS we are referring to. ARNs have a hierarchical structure:

```
arn:partition:service:region:account-id:resource-id
arn:partition:service:region:account-id:resource-type/resource-id
arn:partition:service:region:account-id:resource-type:resource-id
```

- arn: literally, the string "arn"
- `partition` is one of the three partitions: AWS Regions, AWS China Regions, or AWS GovCloud (US) Regions
- `service` is the specific service such as Amazon EC2 or DynamoDB
- `region` is the AWS region like `us-east-1` (North Virginia)
- `account-id` is the AWS account ID
- `resource-id` is the unique resource ID. Other forms for resource IDs like `resource-type/resource-id`, are used by services like IAM where IAM users have `resource-type` of `user` and `resource-id` a username like `MyUsername`,

Try to identify the service, region, and resource for the following example ARNs:

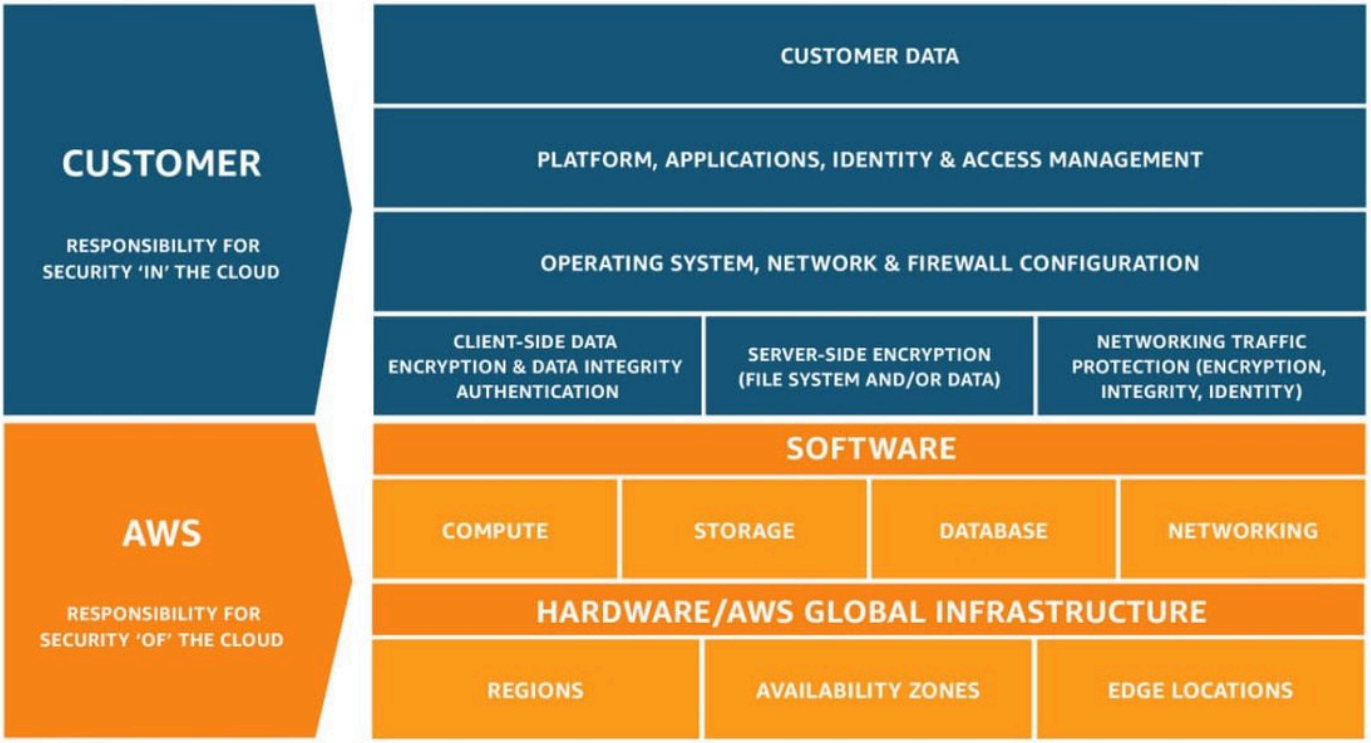
```
arn:aws::dynamodb:us-west-2:123456789012:table/myDynamoDBTable
arn:aws::lambda:us-east-2:123456789012:function:my-function:1
```

If you are interested in learning more, check out a map of Regions and Availability Zones, a view of our data centers, and the complete list of regional service endpoints.

Security model

Security is a top priority for AWS. Before you start building serverless solutions, you need to know how security factors into AWS solutions.

Amazon Web Services has a *shared responsibility model:*



A shared security model means that Amazon manages certain aspects of security, and you are responsible for others.

- AWS is responsible for the security *of* the cloud. This includes such thing as the physical security of the data centers.
- You are responsible for the security *in* the cloud. For example, you are responsible for your data, and for granting functions only the access the need to complete their work.

For developers getting started with serverless and experts alike, the responsibility of securing the resources and functions will take effort to understand and implement.

We'll explain more along the way, but you should at least know that AWS commitment to security is not taken lightly. AWS services have carefully established security mechanisms for you to create secure solutions from the start. However, you have the responsibility to learn how and properly implement these mechanisms in your solutions.

For more details, see the Shared Responsibility Model, AWS Cloud Security, and Security Documentation Index with links to security documents for every service.

Understanding the difference between traditional and serverless development

As a developer, you might already be familiar with traditional web applications, but new to serverless development. You might even know how to use some AWS services directly, such as Amazon S3 or DynamoDB, but are not sure what it takes to develop a fully-functional serverless application in production.

This topic provides an overview a traditional application development, then explain the shift in thinking needed shift to serverless development. This is intended to provide you with a clearer conceptual understanding of serverless development using AWS services

Topics

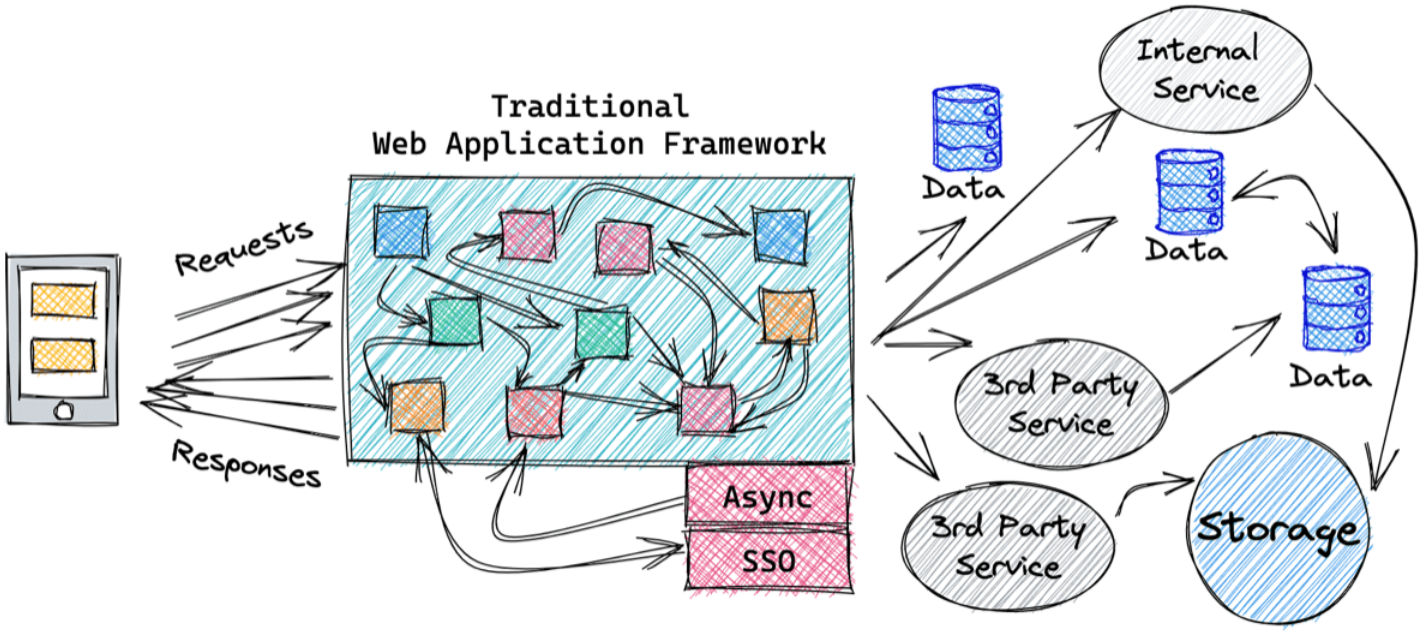
- Traditional development
- Use services instead of custom code
- Serverless development on AWS

Traditional development

Traditional web apps generally handle synchronous requests and responses. This cycle has been the basis of web since the beginning of the Internet. Over time, developers created and shared code to speed up development. You have probably used or at least recognize one or more of these web frameworks: Express, Django, Flask, Ruby on Rails, Asp.net, Play, Backbone, Angular, Spring Boot, Vapor, to name a few.

Web frameworks help you build solutions faster by including common tools and features.

The following diagram represents some of the complex mix of components that are included with frameworks. Routers send URLs to classes or functions to handle requests and return responses. Utility classes retrieve form data, query strings, headers, and cookies. A bundled abstraction layer stores and retrieves data in SQL or NoSQL databases. Additional components manage connections to external services through synchronous API calls or asynchronous message queues. Extension points exist to bolt-on even more components, such as asynchronous hooks, or single sign-on authentication.



We call these solutions *traditional* because request/response has been the model for web applications for decades. We call them *monolithic* because everything is provided in one package.

Traditional frameworks do an awful lot, but can be awfully complex doing it.

To be fair, traditional development does have advantages. Developer workstation setup is generally quick. You typically setup a package manager or installer such as NPM, Gradle, Maven, homebrew, or a custom CLI to initialize a new application, then you run the bare bones app with a command.

Frameworks that bring everything can boost initial productivity, especially when starting with a simple solution.

But, this everything in one box approach makes scaling and troubleshooting problems difficult. As an application grows, it tends to rely on more external systems. Asynchronous integrations with these systems are preferred because they do not block the flow. But, asynchronous requests are difficult to invoke and troubleshoot in a traditional architecture.

For asynchronous actions, the application logic must include timeouts, retry logic, and the status. Single errors can cascade and impact many components. Work flows become more involved just to keep the solution running.

Increases (or decreases) in demand for a particular feature require scaling up (or down) the entire system. This is inefficient and difficult because all of the components and state are tightly coupled together.

The architecture of traditional monolithic web applications tends to become more complex over time. Complexity increases ramp-up time for new developers, makes tracking down the source of bugs more challenging, and delays the delivery of new features.

Use services instead of custom code

Serverless applications usually comprise several AWS services, integrated with custom code run in Lambda functions. While Lambda can be integrated with most AWS services, the services most commonly used in serverless applications are:

Commonly used AWS services in serverless applications

Category	AWS service
Compute	Lambda
Data storage	Amazon S3, DynamoDB, Amazon RDS
API	API Gateway
Application integration	EventBridge, Amazon SNS, Amazon SQS
Orchestration	Step Functions
Streaming data and analytics	Amazon Data Firehose

There are many well-established, common patterns in distributed architectures that you can build yourself or implement using AWS services. For most customers, there is little commercial value in investing time to develop these patterns from scratch. When your application needs one of these patterns, use the corresponding AWS service:

Common patterns and corresponding AWS services

Pattern	AWS service
Queue	Amazon SQS
Event bus	EventBridge
Publish/subscribe (fan-out)	Amazon SNS

Pattern	AWS service
Orchestration	Step Functions
API	API Gateway
Event streams	Kinesis

These services are designed to integrate with Lambda and you can use infrastructure as code (IaC) to create and discard resources in the services. You can use any of these services via the AWS SDK without needing to install applications or configure servers. Becoming proficient with using these services via code in your Lambda functions is an important step to producing well-designed serverless applications.

Serverless development on AWS

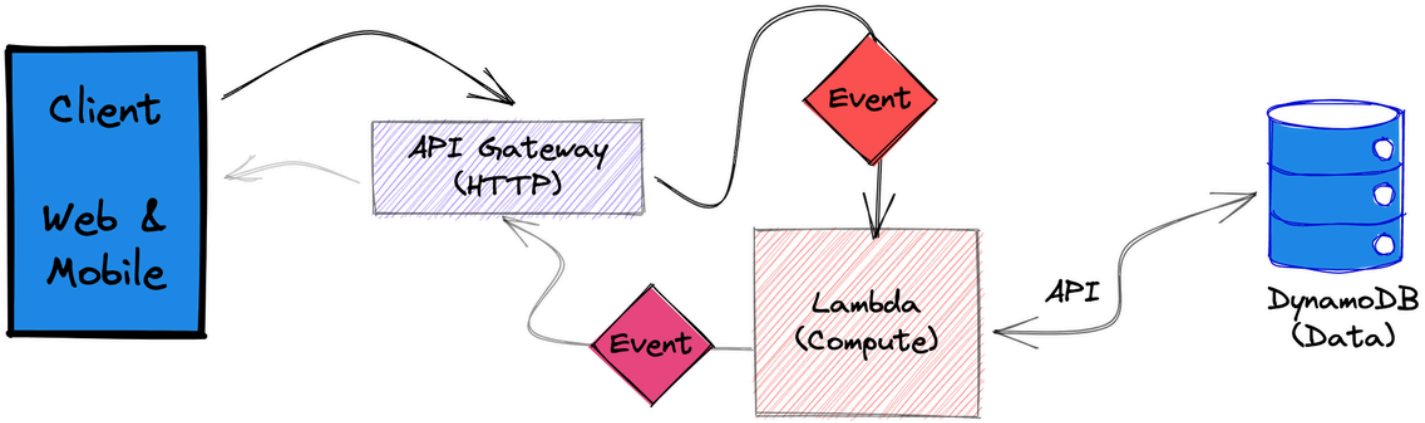
To build serverless solutions, you need to shift your mindset to break up monoliths into loosely connected services. Consider how each service will do one thing well, with as few dependencies as possible.

You may have created microservices before, but it was probably inside a traditional framework. Imagine if your microservice existed, but without the framework. For that to happen, services need a way to get input, communicate with other services, and send outputs or errors.

The key to serverless apps is *event-driven architecture.*

Event-driven architecture (EDA) is a modern architecture pattern built from small, decoupled services that publish, consume, or route *events.* Events are messages sent between services. This architecture makes it easier to scale, update, and independently deploy separate components of a system.

The following diagram shows an event-driven serverless microservice. A client request is converted by an API Gateway into an event that is sent to a Lambda compute service. A Lambda function retrieves info from a DynamoDB data store. That data is returned in an event to API Gateway, which sends a response to the client with all the appropriate headers, cookies, and security tokens.



Many traditional systems are designed to run periodically and process batches of transactions that have built up over time. For example, a banking application may run every hour to process ATM transactions into central ledgers. In Lambda-based applications, the custom processing should be triggered by every event, allowing the service to scale up concurrency as needed, to provide near-real time processing of transactions.

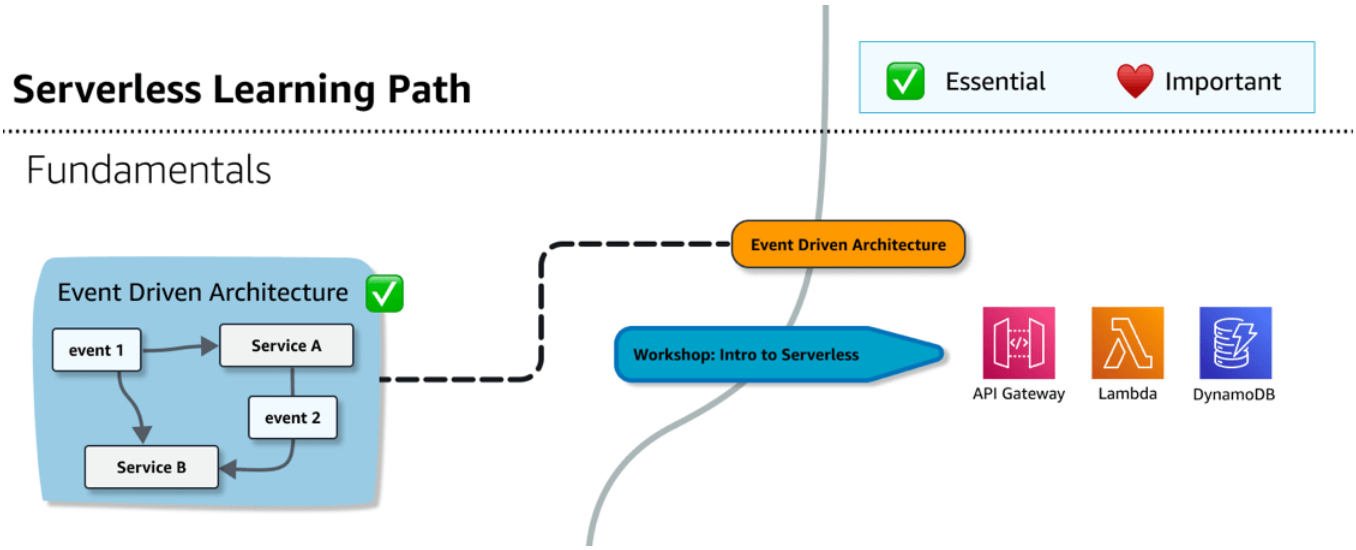
While you can run cron tasks in serverless applications by using Amazon EventBridge Scheduler, consider the size of each batch of data that your event sends to Lambda. In this scenario, there is potential for the volume of transactions to grow beyond what can be processed within the 15-minute Lambda timeout. If the limitations of external systems force you to use a scheduler, you should generally schedule for the shortest reasonable recurring time period.

For example, it's not best practice to use a batch process that triggers a Lambda function to fetch a list of new Amazon S3 objects. This is because the service might receive more new objects in between batches than can be processed within a 15-minute Lambda function.

Transitioning to event-driven architecture

Event-driven architecture (EDA) is the first step on the serverless learning path. Understanding how services interact through *events* is essential to successful serverless development.

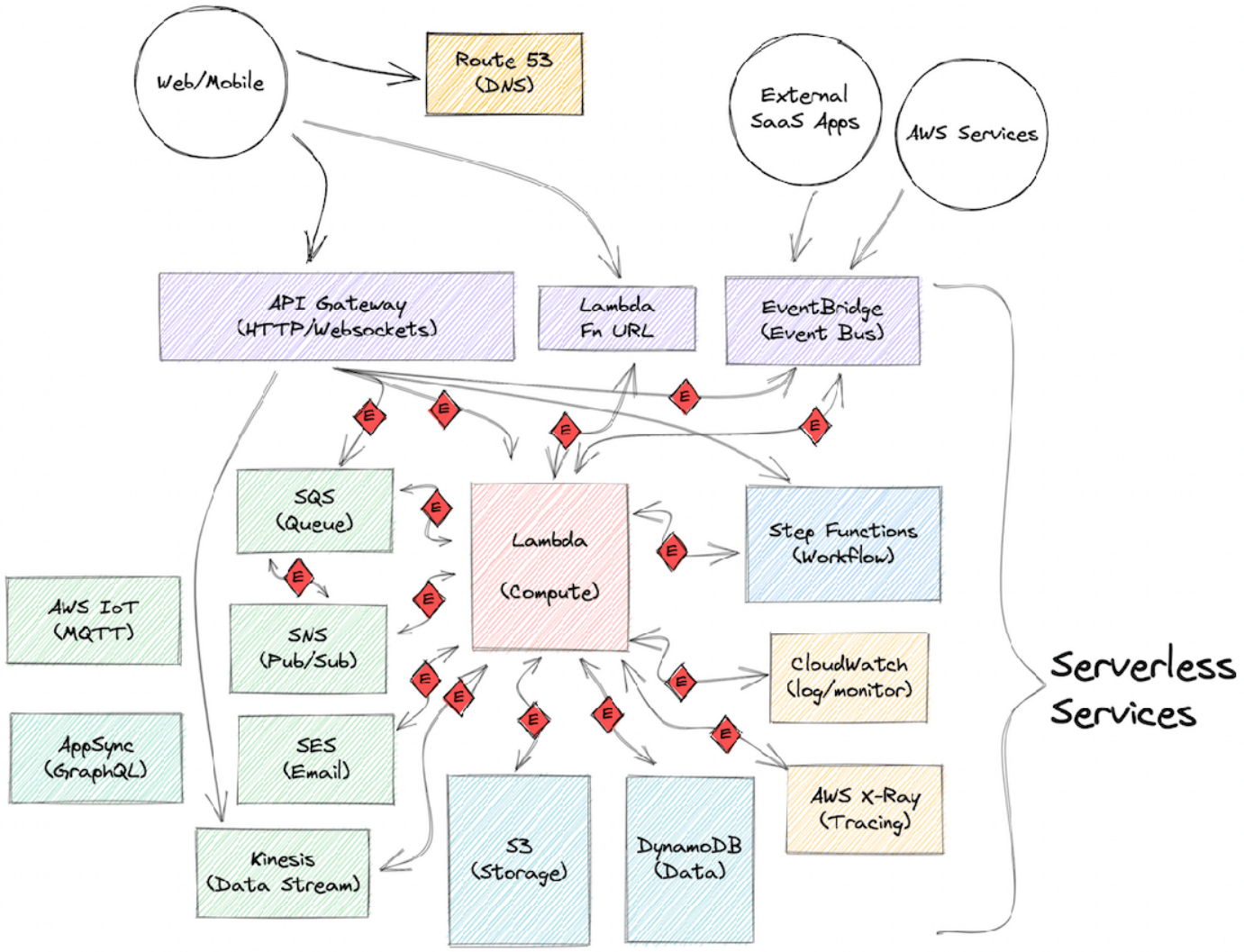
In this chapter, you will dive into the transition from traditional to event-driven architecture.



Let's start by thinking about event-driven components of a food delivery service.

You'll need a data store for menus, locations, orders, status. Customers will send network requests to a web API. Your application then needs a compute resource to process customer orders.

You can handle long-running tasks asynchronously. For example you can implement a queue using Amazon SQS to manage order submission on. You can then use Step Functions to manage a workflow that updates user information, and inventory counts after every order is processed. Along the way, you will need to log actions, monitor app activity, and trace data flows to debug.

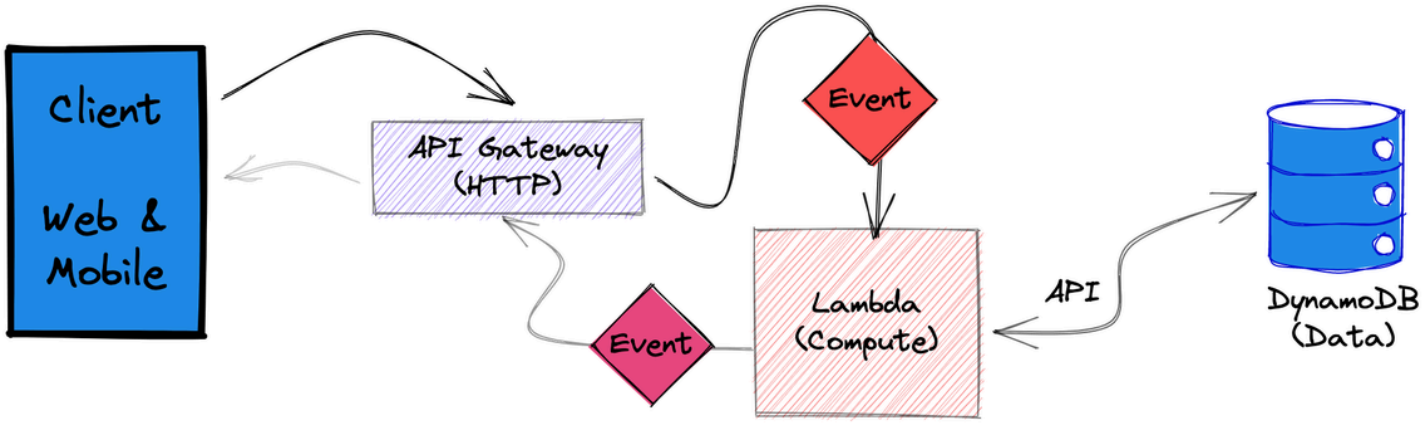


If you're new to serverless development, you might be more familiar with traditional frameworks. Let us look at the steps in a traditional request/response cycle for comparison:

1. Accept an inbound network request and create local data objects.
2. Map the URL, often called a route, with configuration or annotations to an action.
3. Create global data and utility services, such as a database connection pool and an object relational mapper.
4. Implement web hooks for logging, observability, and health monitoring components.
5. Process the request: query a database, store or retrieve data, call external systems.
6. Convert outbound data into a suitable response or error and serialize the data into JSON.
7. Add metadata, such as headers, cookies, tokens to the response object.
8. Send the response back to the client.

Now, let us compare a similar work flow implemented with event-driven architecture.

This diagram represents a microservice that retrieves data from a database, for example, retrieving shopping cart items for a customer order.



First, a web or mobile client makes an HTTP request to `GET /cart/A1234B56` for a list items in a cart.

A component needs to accept the request and extract metadata, such as HTTP method, path, extra path info, query string parameters, headers, and cookies. That component will also verify the request is from an authenticated and authorized entity. In AWS, API Gateway accepts the inbound URL, extracts the parameters, query string, and headers and creates an event to send to other services for processing.

Example based on event.json API Gateway proxy event for a REST API:

```
{
  "resource": "/cart",
  "path": "/cart/A1234B56",
  "httpMethod": "GET",
  "headers": {
      "accept": "text/html,application/xhtml+xml,application/xml;q=0.9",
      "accept-encoding": "gzip, deflate, br",
      "User-Agent": "Chrome/80.0.3987.132 Safari/537.36",
      "X-Amzn-Trace-Id": "Root=1-5e66d96f-7491f09x5pl79d18acf3d050", ...
  },
  "multiValueHeaders": {
      "accept": [
          "text/html,application/xhtml+xml,application/xml;q=0.9,"
      ],
```

```
      "accept-encoding": [
          "gzip, deflate, br"
      ],

      ...

      "queryStringParameters": null,
      "multiValueQueryStringParameters": null,
      "pathParameters": null,
      "stageVariables": null,
      "body": null,
      "isBase64Encoded": false
  }
}
```

Tip

A dedicated API service might at first seem unnecessary, but implementing this action as a separate service allows flexibility and scalability to your solution. For more information about working with API Gateway, see the the section called “API Gateway”.

An *event* represents a change in state, or an update.

For example: an item placed in a shopping cart, a file uploaded to a storage system, or an order becoming ready to ship. Events can either carry the state, such as: quantity (qty), item price (itemPx), and currency; or simply contain *identifiers* needed to look up related information, such as: customerId and orderId, as shown in the following example of a `NewOrderEvent`:

Event name

NewOrderEvent

```
{
  "source": "myApplication",
  "detail": "submitOrder",
  "customerId": "customer123",
  "orderId": "order-A1234B56",
  "paymentStatus": "open",
  "cart": [
    {
      "product12": {
        "qty": 2,
        "itemPx": 35.22,
        "currency": "USD"
      },
      "product44": {
        "qty": 5,
        "itemPx": 71.57,
        "currency": "USD"
      }
    }
  ],
  "timestamp": 1607774286
}
```

Next, API Gateway integrates with Lambda, a compute service, to handle the new event. Lambda function code parses the parameters in the inbound event, connects to the data store, and retrieves the cart. The function queries the database API through an SDK library. Because the DynamoDB database is also serverless and built to respond with low latency, there is no need for a connection pool.

After converting currency to USD and removing unavailable items, the function sends the result as a new event to API Gateway.

Finally, API Gateway converts the event into a response to send to the waiting client.

The method with which a function is invoked should be informed by your application archecture and needs. For example, batch-processing patterns have different applications to on-demand data processing. Understanding these paradigm differences can also help customers decide between AWS services.

Deploying a microservice as a containerized application on Fargate could be more appropriate if the microservice is primarily used for batch data processing. Whereas a Lambda function would be much more straight-forward to deploy and maintain in applications that require on-demand data processing.

Decoupled event-driven architecture

For simpler applications, the advantage of event-driven versus request-driven applications may not be apparent. But, as your applications add more functionality and handle more traffic, the value becomes clear.

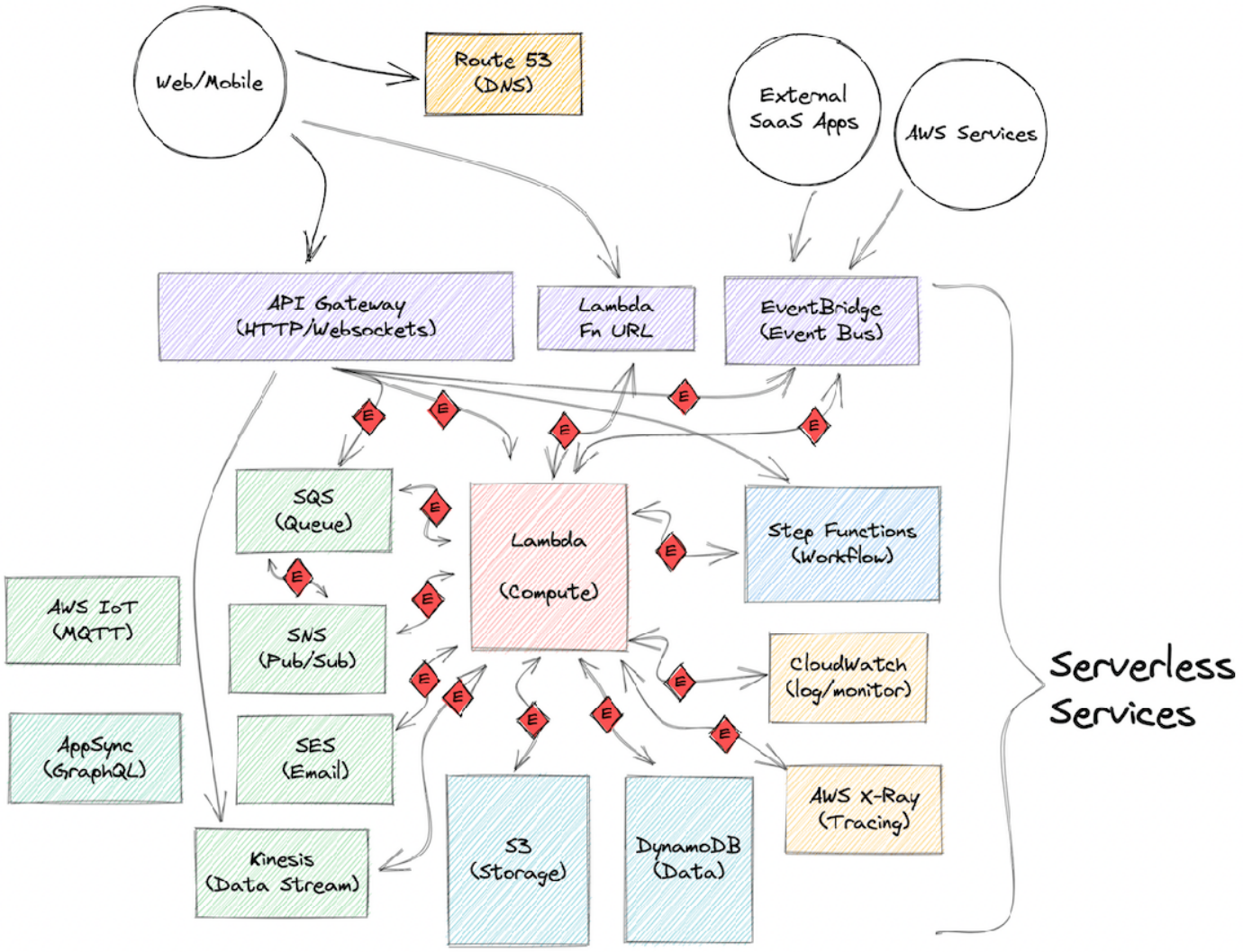
Event-driven applications rely on communication through events that are also observable by other services and systems. Event producers are unaware of which (if any) consumers are listening. This strategy makes it easier to extend and scale, without disrupting existing work flows.

For example, API Gateway can enforce rate and volume limits for requests to your API on a per-customer basis. Or, a service could watch the inbound query parameters to create a list of popular product searches. The database could send events when products are added to the cart, to feed to a predictive ML algorithm for ordering supplies.

Due to the loose coupling between components of an event-driven system, your compute functions are not even aware of these other activities. You can scale components independently. One service can fail, without impacting other services. Events can be flexibly routed, buffered, and provide a log for audit.

Let's revisit the diagram with various services connected through events. We can start to see now that it is actually not as complex as it might have appeared.

Think of it like looking down on a big city with messengers moving packages and letters between people and businesses. The sources and destinations range from the suburbs to the city core. Inbound requests could be managed by a dispatcher, like API Gateway, Lambda Function URLs, or Amazon EventBridge. Or, some messages might be dropped in a box for asynchronous delivery. This would be like events routed to queues or orchestrated in complex work flows with Step Functions. The monitoring, tracing, and metrics services, CloudWatch and AWS X-Ray, are like managers, watching the a stream of events to make sure packages are delivered, and if not, to troubleshoot the problem. After drop-off at a business, some packages are transferred by in-house delivery agents to their final destination. This situation is similar to how services that store data and files, may stream events to compute services, which triggers ever more actions.



Connecting services with **events** and **event-driven architecture** gives you a consistent and scalable way to build solutions with hundreds of services.

Summary

- Serverless is built on independent services that communicate through *events* in an *event-driven architecture.*
- *event-driven architecture* (EDA) - a modern architecture pattern built from small decoupled services that publish, consume, or route *events.*
- *events* - represent a change in state, or an update
- Decoupled microservice architecture helps you build modern, agile, and extendable applications faster than traditional monolithic applications, and free developers from needing to learn everything at once about existing systems.

Next steps

- See What is EDA? for advantages of a decoupled architecture.
- Learn more about the advantages of modernizing monolithic applications in the AWS Prescriptive Guidance enabling data persistence in microservices reference document.

Focusing on core serverless services

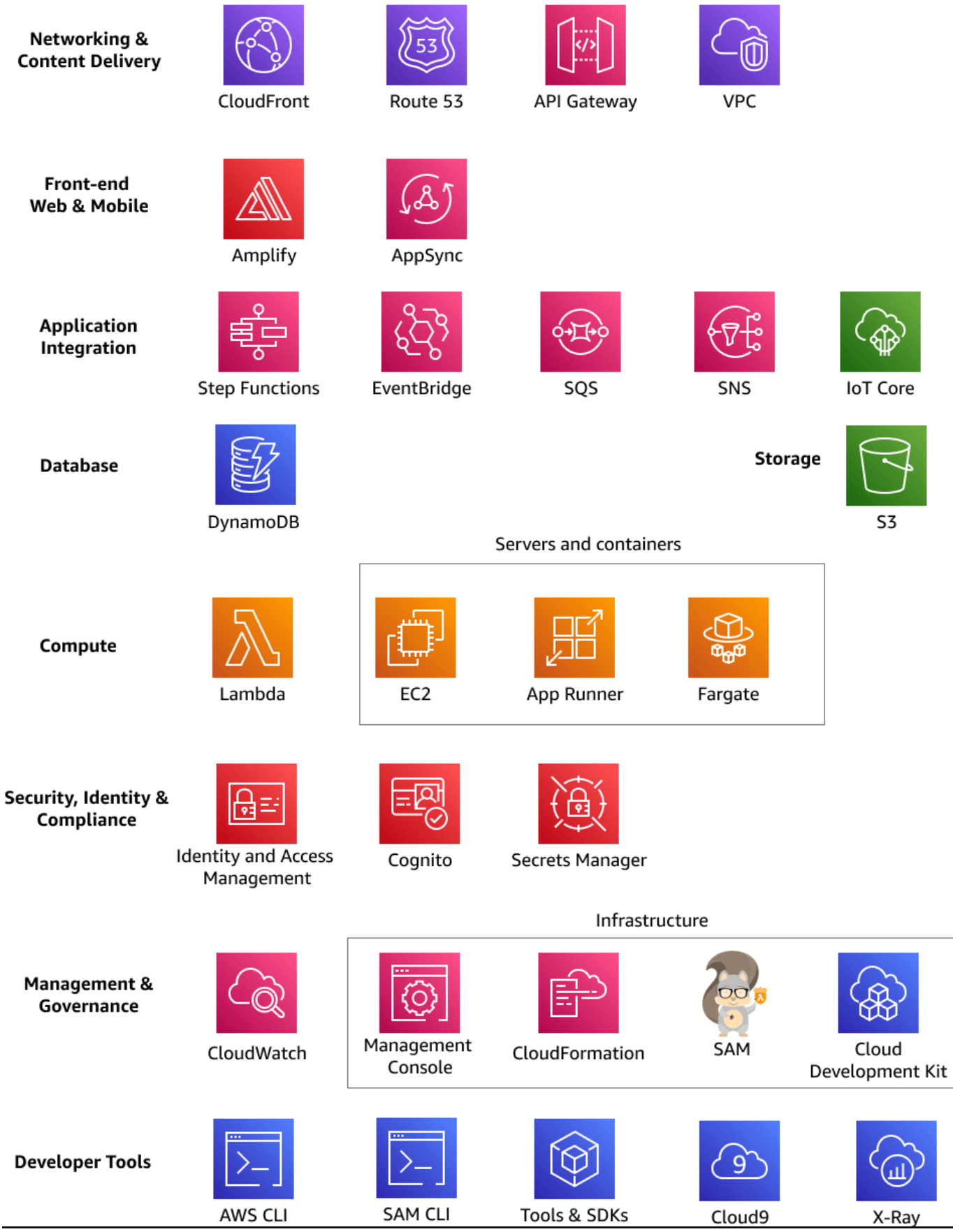
AWS has over 220 services.

Each service is a tool in your serverless development toolbox. Commonly, you start out using some services more frequently than others. This topic provides an overview of the core services you need to build serverless solutions.

You can read high level explanations of the core services here, and an example of how they interact within the context of an example microservice, or you can choose to skip ahead to the hands on workshop that uses three common services to build a working microservice.

Common serverless services

The following diagram shows AWS services commonly used together to build serverless applications:


Networking & Content Delivery
CloudFront
Route 53
API Gateway
VPC
Front-end Web & Mobile
Amplify
AppSync
Application Integration
Step Functions
EventBridge
SQS
SNS
IoT Core
Database
DynamoDB
Storage
S3
Servers and containers
Compute
Lambda
EC2
App Runner
Fargate
Security, Identity & Compliance
Identity and Access Management
Cognito
Secrets Manager
Infrastructure
Management & Governance
CloudWatch
Management Console
CloudFormation
SAM
Cloud Development Kit
Developer Tools
AWS CLI
SAM CLI
Tools & SDKs
Cloud9
X-Ray

Networking & content delivery

- **Amazon CloudFront** - content delivery network, serving and caching assets in storage
- **Amazon Route 53** - DNS registry/service
- **Amazon API Gateway** - HTTP & WebSocket connections and integrations
- **Amazon Virtual Private Cloud** - private networking between services in the cloud

Front-end web & mobile

- **AWS Amplify** - open-source client libraries to build cloud powered mobile and web apps on AWS with authentication, data store, pub/sub, push notifications, storage, API built on AppSync
- **AWS AppSync** - managed GraphQL API

Application integration

- **AWS Step Functions** - orchestration service; useful when you have workflows with more than one state, need to branch, or run tasks in parallel. The Step Functions service acts as the state model for your application.
- **Amazon EventBridge** - integration with AWS & 3rd party services through events
- **Amazon Simple Queue Service** - simple queue service; buffering requests
- **Amazon Simple Notification Service** - simple notification system, publish/subscribe topics, and sending a limited number of SMS/email messages
- **AWS IoT Core** - bi-directional communication for Internet-connected devices (such as sensors, actuators, embedded devices, wireless devices, and smart appliances) to connect to the AWS Cloud over MQTT, HTTPS, and LoRaWAN
- **Amazon Simple Email Service** - simple email system, bulk email sending service

Database & storage

- **Amazon DynamoDB** - scalable no SQL key/value store
- **Amazon Simple Storage Service** - file storage

Compute

- **AWS Lambda** - serverless compute functions; responsible for nearly all processing in serverless projects
- **Amazon Elastic Compute Cloud** - non-serverless compute alternative; useful when you need always-on and fully customizable capabilities. EC2 is often used for initial “lift and shift” migration to the cloud. You can continue to use EC2 while migrating portions of your workflow to serverless patterns.
- **AWS App Runner** - fully managed service to deploy your containerized web applications and APIs. App Runner will scale compute instances and network resources automatically based on incoming traffic.
- **AWS Fargate** - serverless computer for clusters of containers; useful when you need custom containers but do not want to maintain and manage the infrastructure or cluster.

Security, identity & compliance

- **IAM** - identity and access management; provides policies to authorize service resources to interact with each other and your data.
- **Amazon Cognito** - authentication and authorization of users and systems
- **AWS Secrets Manager** - manage access to secrets using fine-grained policies

Management & governance

- **Amazon CloudWatch** - suite of monitoring and logging services
- **AWS Management Console** - web-based user interface for creating, configuring, and monitoring AWS resources and your code.
- **AWS CloudFormation (CFN)** - text templates to automate deploying infrastructure and code
- **AWS Serverless Application Model (AWS SAM)** - an open-source framework for deploying serverless application infrastructure and code. AWS SAM templates provide a shorthand syntax to declare functions, APIs, databases, and event source mappings. With just a few lines of configuration per resource, you can define the application infrastructure components. During deployment, AWS SAM transforms and expands the template into verbose AWS CloudFormation templates.
- **AWS Cloud Development Kit (AWS CDK)** - an open-source software development framework to define your cloud application resources using familiar programming languages. Instead of

configuration files, you write code that creates infrastructure. Your IDE can validate the definition and even provide hints through code completion.

Developer tools and code instrumentation

- **AWS CLI** - command line utility for managing AWS resources
- **AWS SAM CLI** - command line utility for rapidly creating, deploying, and testing AWS resources with AWS SAM templates
- **Tools & SDKs** - libraries for connecting to services and resources programmatically
- **Cloud9** - cloud-based integrated development environment
- **X-Ray** — monitoring and debug

Streaming & batch processing

- **Kinesis** - event stream processing at scale

Typical microservice example

Consider the following scenario: you want to build a microservices application that looks up weather data by zip code and returns JSON data.

What serverless services would you use, and how?

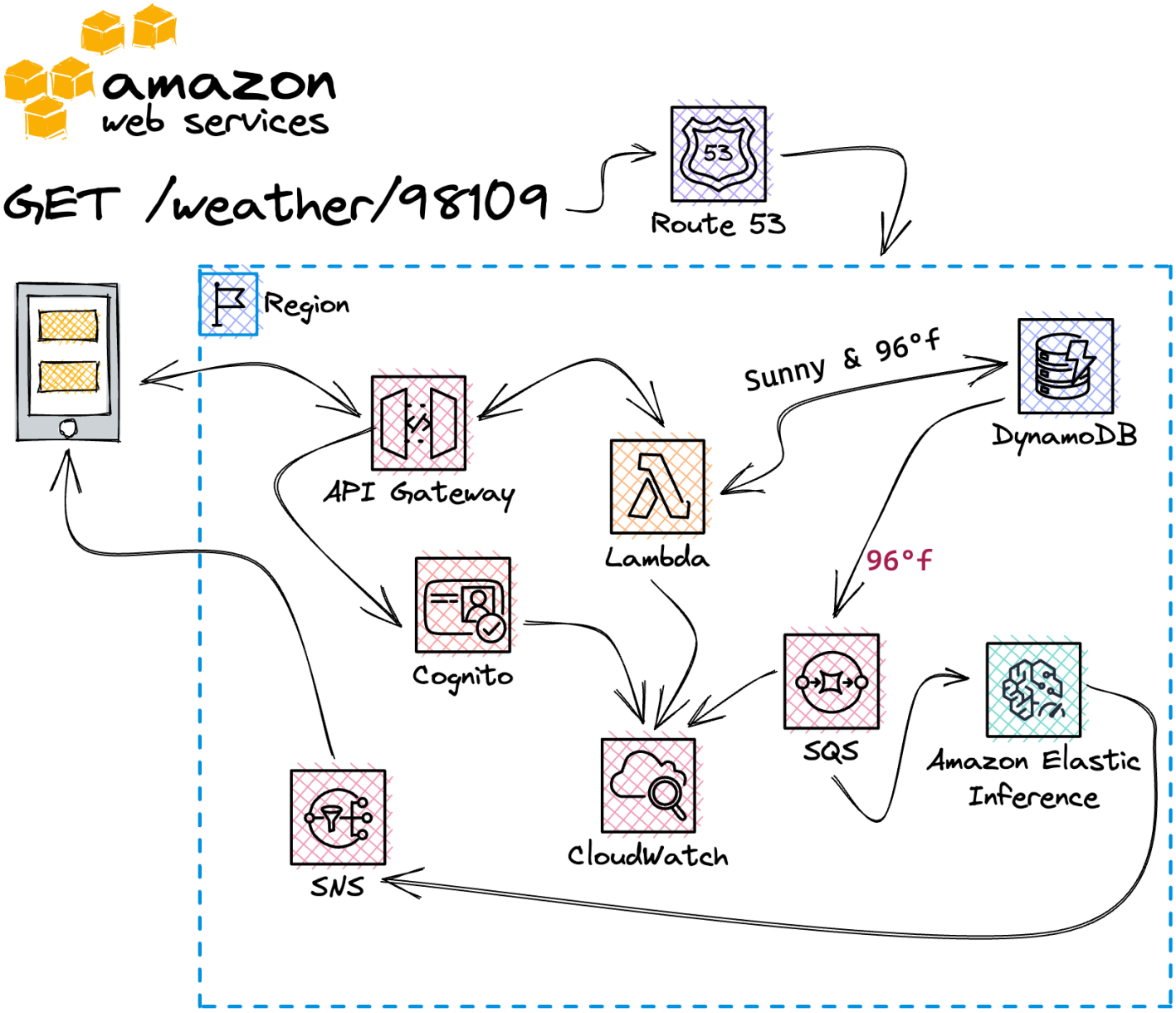
The solution starts with the client resolving the hostname through Route 53 DNS. The browser's HTTPS GET request routes to API Gateway. If the URL is valid, API Gateway verifies access an access token, commonly implemented as a JWT tokenJWT token with Amazon Cognito, then creates an event for the request and sends it to a serverless Lambda function for processing.

The Lambda function receives the event and a *context* object with additional information related to the environment as inputs to a designated *handler* method. The handler method in this case, uses an SDK to send a query to DynamoDB for weather data for the given zip code. The function may filter and customize the data based on the location and preferences of the user, perhaps converting degrees in Celsius to Fahrenheit.

Before returning the data, bundled into a new event, back to API Gateway, the function handler might create additional events. It might send one to an SQS queue, where a data analytics service

could be listening. The handler function might create and send another event to an SNS queue so that alerts for high temperature are sent to users through SMS messages.

The function finally wraps up the JSON weather data into a new event and sends it back to API gateway. Afterward, the function continues to handle hundreds of additional requests. Request from users slow down after 2AM, so after some time the Lambda service will tear down the function execution environment to conserve resources. As a Customer, you will only be charged for function usage.



Getting started with serverless applications

Core *service starters* will quickly explain the value and technical fundamentals of each service. Each starter will also mention advanced topics, so you can start with the essentials, but be aware of capabilities to dive into when you need them.

Starters are short reads (less than 2,300 words; 10-15 min) that connect concepts and practical hands-on use.

Topics

- Get started with IAM
- Get started with Lambda
- Get started with API Gateway
- Get started with DynamoDB
- Learn using a workshop

Get started with IAM

Interactions with AWS services and resources by developers and entities require:

- **Authentication**: proof that the entity requesting access is who they claim to be
- **Authorization**: actions that are allowed or denied

What is Identity and Access Management?

AWS provides and uses a service called Identity and Access Management (IAM) for authentication and authorization. IAM is used to manage developer accounts and secure the interaction between services and resources.

> **Warning**
>
> Security is an important, complex, and broad topic. Large organizations generally have specific operational procedures that developers need to follow. This guide will explain only essential concepts necessary to get started with AWS services. If in doubt, consult your IT department or the official security documentation.

Fundamentals

With IAM, developers attach *policies,* JSON documents that define granular permissions, to resources. IAM provides pre-built AWS managed policies for common access levels. You can also define your own policies with the least-privilege level necessary to complete tasks.

Information about IAM policies may come at you fast. If it gets to be too much, put it in **PARC**:

- **P**rincipal: entity that is allowed or denied access
- **A**ction: type of access that is allowed or denied
- **R**esource: AWS resources the action will act upon
- **C**ondition: conditions for which the access is valid

At a high level, these four terms should be enough to get you started connecting serverless resources.

Account prerequisites

But, before you start, you need an AWS account. The following sections provide the best practice steps to create an account and an administrative user.

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

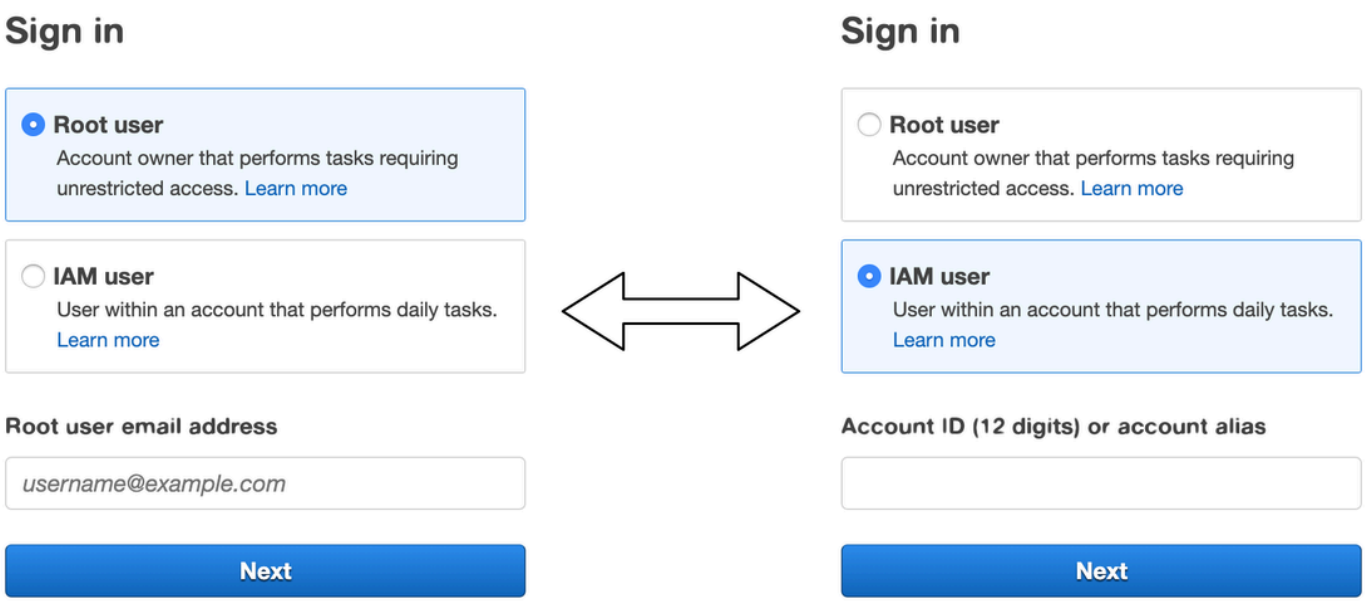
1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.

2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

A common confusion arises when signing in to AWS. Remember, for day to day activities, you should **not** be signing in as the root user.



Principals

IAM implements *authentication*, proving who an entity claims to be, with *principals,* which are entities such as IAM users, federated users from Google, Facebook, etc, IAM roles, AWS accounts, and AWS services.

> **Tip**
>
> An IAM role is identical in function to an IAM user, with the important distinction that it is not uniquely associated with one entity, but assumable by many entities. Typically, IAM roles correspond to a job function.

A loose analogy for IAM roles are that of professional uniforms: a surgeon's scrubs, a firefighter's hardhat, or a startup CTO's favorite hoodie. Many people can *assume the role* of a surgeon, firefighter, and startup CTO, which identifies them with a certain job function.

One of the most useful things about IAM roles is they can be associated not only with human entities, but also with AWS services. These types of roles are known as *service roles*. This means you can assign an IAM role directly to a service. With an IAM role assigned to the service instance, you can then associate specific IAM policies with the instance role, so that the service instance itself can access other AWS services. This is extremely useful for automation.

Authorization - PARC

So far we've been talking about principals. Principals represent the **authentication** component. For authorization, you will attach JSON documents called *IAM policies* to principals.

Principals

As mentioned, *principals* are the entities that are allowed or denied access.

Actions

Actions are the type of access that is allowed or denied. Actions are commonly AWS service API calls that represent create, read, describe, list, update, and delete semantics.

Resources

Resources are the AWS resources the action will act upon.

All AWS resources are identified by an Amazon Resource Name (ARN) . Because AWS services are deployed all over the world, ARNs function like an addressing system to precisely locate a specific component. ARNs have hierarchical structures:

```
arn:partition:service:region:account-id:resource-id
```

```
arn:partition:service:region:account-id:resource-type/resource-id
arn:partition:service:region:account-id:resource-type:resource-id
```

- `arn` means this string is an ARN
- `partition` is one of the three AWS partitions: AWS regions, AWS China regions, or AWS GovCloud (US) regions
- `service` is the specific AWS service, for example: EC2
- `region` is the AWS region, for example: us-east-1 (North Virginia)
- `account-id` is the AWS account ID
- `resource-id` is the unique resource ID. (Could also be in the form `resource-type/resource-id`)

Related resource(s):

- IAM identifiers provides an exhaustive list in the docs for IAM ARNs

Conditions

Conditions are specific rules for which the access is valid.

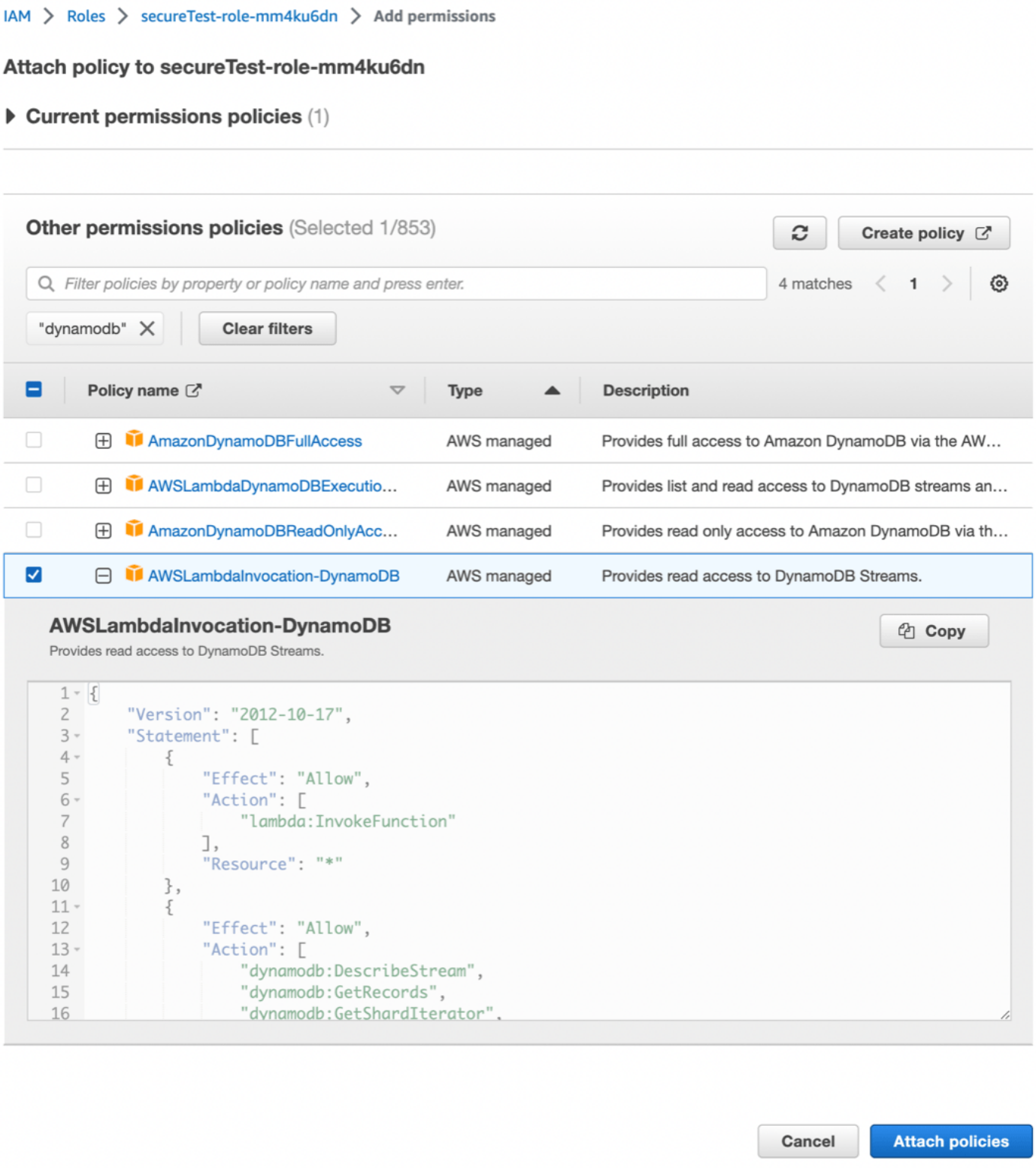
Other Elements

- All IAM policies have an *Effect* field which is set to either `Allow` or `Deny`.
- `Version` field defines which IAM service API version to use when evaluating the policy.

- `Statement` field consists of one or many JSON objects that contain the specific Action, Effect, Resource, and Condition fields described previously
- `Sid` (statement ID) is an optional identifier for a policy statement; some services like Amazon Simple Queue Service and Amazon Simple Notification Service might require this element and have uniqueness requirements for it

Policies

When you set permissions, you attach a JSON policy to a principal. In the following example, an AWS managed policy named **AWSLambdaInvocation-DynamoDB** will be attached to a role that is related to a Lambda function:



You can also create customer policies that apply to specific actions, resources, and custom conditions. The following policy example includes a condition that applies to an IP range:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeInstances",
                "ec2:RunInstances"
                ...
                <additional actions>
            ],
            "Resource": "*",
            "Condition": {
                "IpAddress": { "aws:SourceIp": "12.34.56.78/32"}
            }
        }
    ]
}
```

First, notice there is no `Principal` element in the policy. This is because attaching a policy to a principal implicitly specifies the principal to which the policy applies. Policies can exist apart from principals, so that common policies can be re-used for many roles, services, etc.

The `Effect` is `Allow`, so this is a policy that explicitly grants access. This is the most common type of policy. You can write policies that explicitly `Deny` access as well.

The `Action` is an array of multiple actions that permit using EC2 to run instances and describe instance metadata. Actions can be a single action, or multiple actions in an array like depicted here.

There is no specific ARN in the `Resource` field, but instead `*`, which is a wildcard character that means this policy applies to all resources.

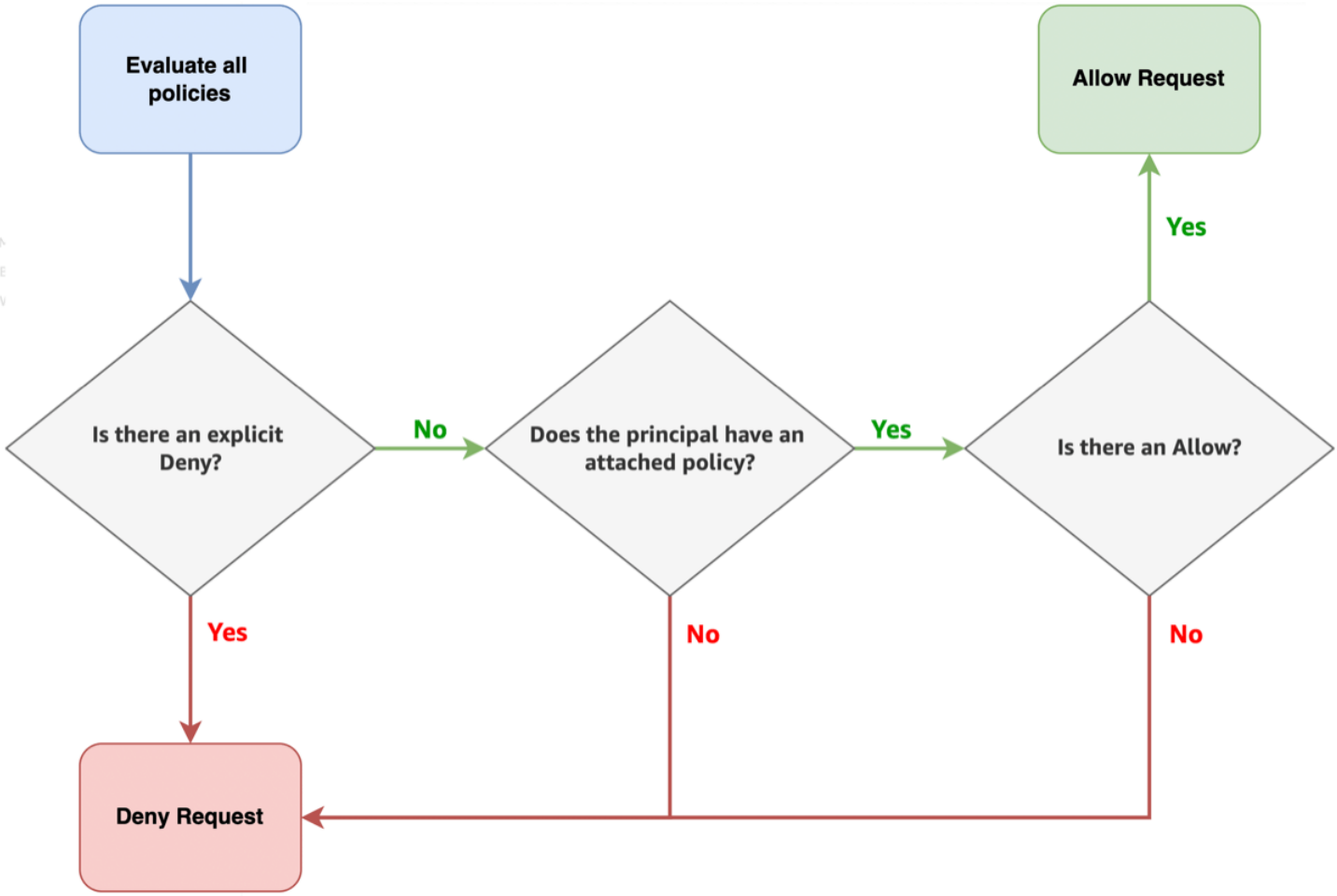
Finally, we have a `Condition` set that applies this policy only if the caller's IP address matches exactly 12.34.56.78. Conditions are optional, and do not need to be specified if the policy is to be applied unconditionally.

IAM policies can be combined, each with varying degrees of sensitivity and specificity. The net effect of combining policies is fine-grained access control for every resource in an AWS account.

How policies are evaluated

For IAM principals, requests to AWS are implicitly ***denied***. This means that if no policies are attached to a principal, IAM's default behavior is to deny access.

Next, if the principal does have an attached policy, and there is an explicit allow, the implicit deny is overridden. However, an explicit deny in any policy overrides any allows. In complex situations, there can be additional steps, but the following diagram represents this simplified model of how IAM evaluates identity based policies:



> **Warning**
>
> Identity based policies do not affect the **root user**, so actions taken by the **root user** account are implicitly ***allowed***.
> The root user is special in this regard and is the **only** principal that has this type of access.

Advanced topics

You can do a lot just using AWS managed policies. As you progress on your journey, you should explore the following more advanced topics.

Resource-based policies

When you create a permissions policy to restrict access to a resource, you can choose an *identity-based policy* or a *resource-based policy*.

Identity-based policies are attached to a user, group, or role. These policies let you specify what that identity can do (its permissions). For example, you can attach the policy to the group named RemoteDataMinders, stating that group members are allowed to get items from an Amazon DynamoDB table named `MyCompany`.

Resource-based policies are attached to a resource. For example, you can attach resource-based policies to Amazon S3 buckets, Amazon Simple Queue Service queues, VPC endpoints, and AWS Key Management Service encryption keys.

With resource-based policies, you can specify who has access to the resource and what actions they can perform on it.

Related resource(s):

- Identity-based and resource-based policies in the official documentation
- AWS Services that work with IAM is a comprehensive list of services, including which ones support resource-based policies

IAM permissions boundaries

With a permissions boundary, you set the maximum permissions that an identity-based policy can grant to an IAM entity.

When you set a permissions boundary for an entity, the entity can perform only the actions that are allowed by both its identity-based policies and its permissions boundaries. Permissions boundaries limit the maximum permissions for the user or role.

For example, assume that the role named `CoreServiceAdmin` should be allowed to manage only Amazon S3, Amazon CloudWatch, and AWS Lambda. To enforce this rule, you can set a policy to set the permissions boundary for the `CoreServiceAdmin` role.

Related resource(s):

- Permissions boundaries for IAM entities - official documentation.

Additional resources

Official AWS documentation:

- AWS Identity and Access Management Documentation
- Example IAM identity-based policies - an extensive list of example policies, including AWS Lambda: Allows a lambda function to access an Amazon DynamoDB table which is useful in microservices
- Grant least privilege section of the *Policies and permissions* chapter suggests a method to refine permissions for increased security

Resources from the serverless community:

- Simplifying serverless permissions with AWSAWS SAM Connectors - AWS Compute blog post by Kurt Tometich, Senior Solutions Architect, AWS, from Oct 2022 that introduces a AWS SAM abstraction that creates minimally scoped IAM policies
- Building AWS Lambda governance and guardrails - AWS Compute blog post by Julian Wood, Senior Solutions Architect, AWS, from Aug 2022 that highlights how Lambda, as a serverless service, simplifies cloud security and compliance so you can concentrate on your business logic.

Next Steps

- Work through the Getting Started Resource Center 30-45 min tutorial on Setting Up Your AWS Environment to properly set up your AWS account, secure the root user, create an IAM user, and setup AWS CLI and (optionally) Cloud9 environment.

Get started with Lambda

All projects need a compute capability to handle processing tasks. Here are some examples:

- Handling web application and API requests
- Transforming batches of data
- Processing messages from a queue

- Resizing images
- Generating dynamic PDFs from customer data

In traditional applications, you write code to do these tasks. You organize that code into **functions**. You put the function code inside an application framework. Whichever framework you picked will run inside a language dependent runtime environment. Finally, that runtime environment will be hosted on a virtual or physical server.

Setting up, configuring and maintaining the frameworks, runtime environments, and virtual or physical infrastructure slows down your delivery of features, bug fixes, and improvements.

What is Lambda?

In Lambda, you write function code. Lambda runs the functions. That's it. There are no servers.

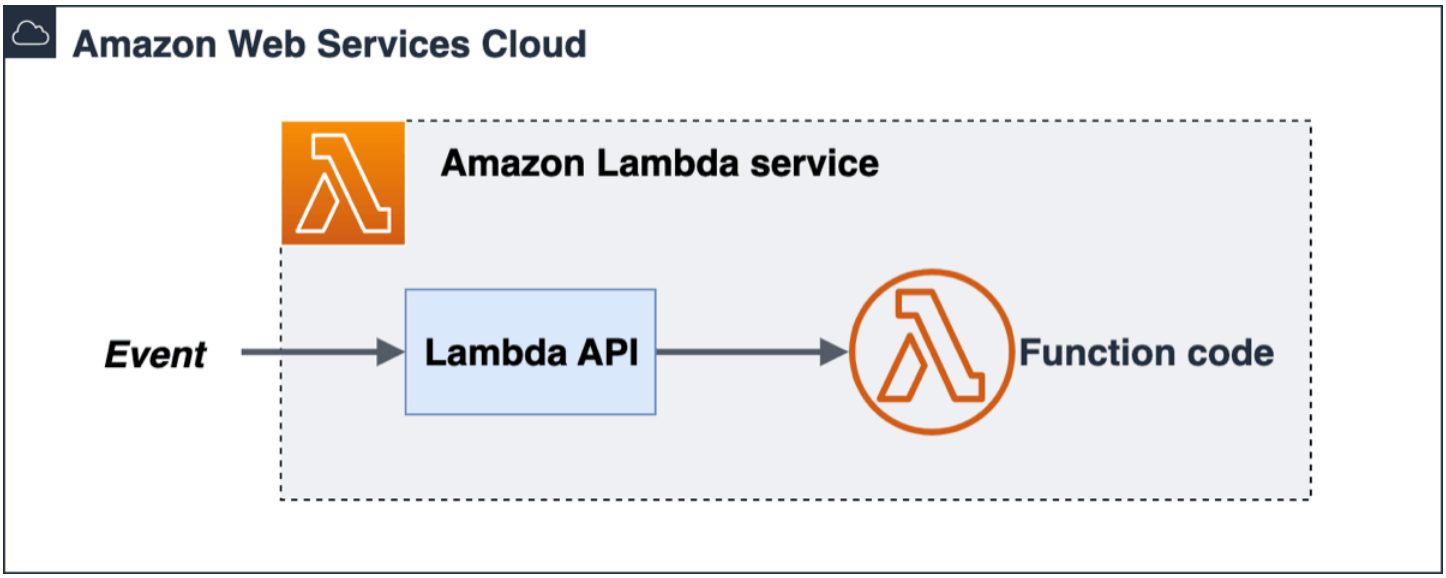
"No Server Is Easier To Manage Than No Server" - Werner Vogels, VP and CTO

The Lambda service runs instances of your function only when needed and scales automatically from zero requests per day to thousands per second. You pay only for the compute time that's actually used — there is no charge when your code is not running.

Fundamentals

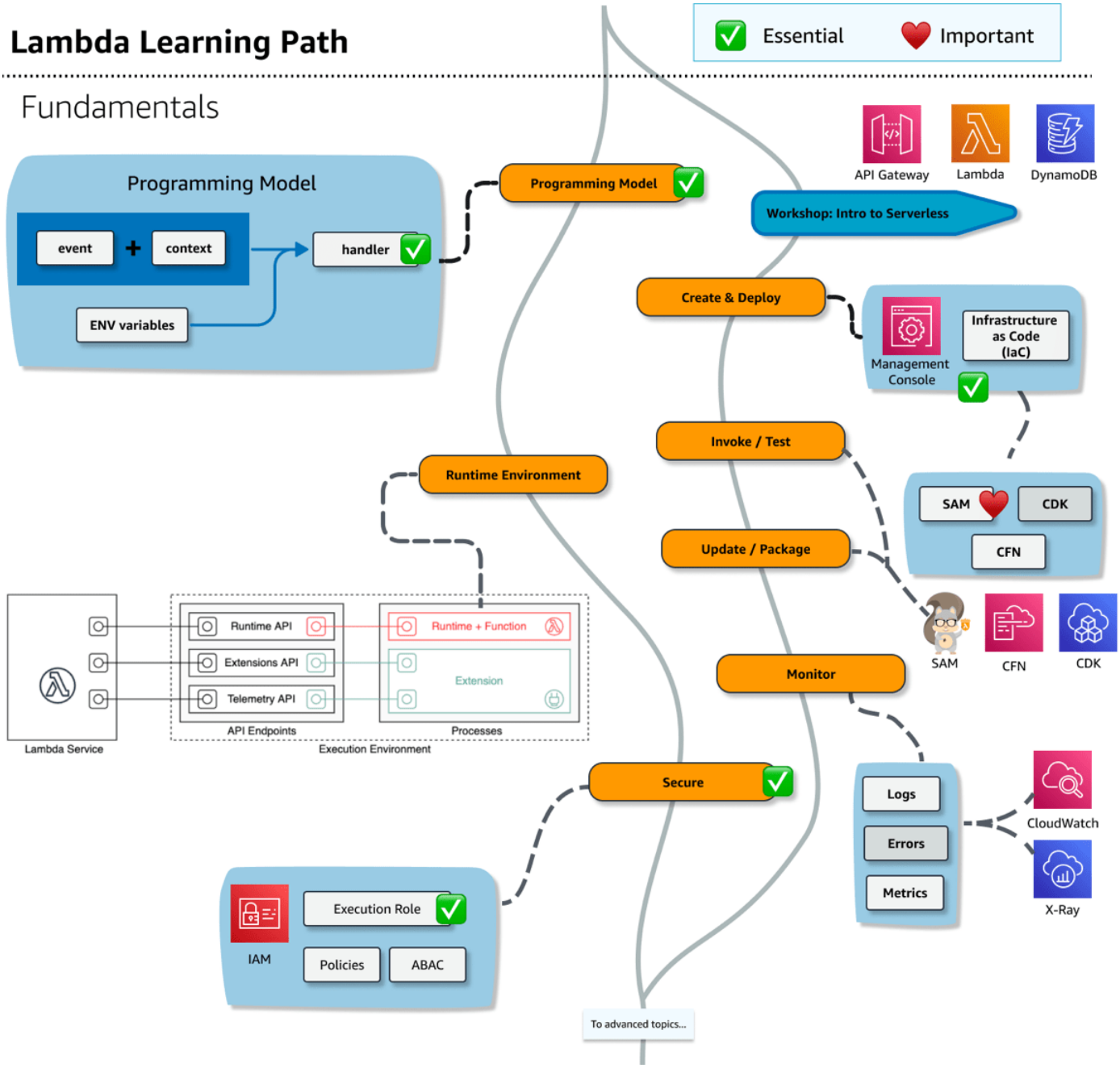
Serverless solutions are based on *event-driven architecture,* or EDA, where services send and receive *events*, which represent an update or change in state. The primary activity of Lambda functions is to process events.

Within the Lambda service, your function code is stored in a code package, deployed as a .zip or a container image. All interaction with the code occurs through the Lambda API. There is no direct invocation of functions from outside of the Lambda service.



What you will learn on your journey to building applications with Lambda:

- How the event-driven programming model invokes Lambda functions
- How to create, invoke, test, update, package, and secure functions
- How the execution and runtime environment runs your functions
- How to view logs and monitor your functions
- Where to find hands-on opportunities to learn how to invoke functions



Fundamentals - conceptual and practical paths

The following is a text representation of the key concepts in the preceding diagram.

The Lambda learning path forks into two paths. The conceptual path focuses on the programming model, runtime environment, and security concepts. The other path includes practical steps to build a application while introducing development workflow activities such as how to create and

deploy functions, invoke and test, update and package, and monitor the logs and troubleshoot errors.

Programming Model

- Event plus Context and Environment variables (ENV) are inputs to a Handler function
- ENV variables
- Runtime environment

Create & Deploy

- Management Console
- Infrastructure as Code (IaC) - AWS CloudFormation (CFN), AWS SAM (SAM), AWS Cloud Development Kit (AWS CDK)
- Deploy .zip file archives — when you need additional libraries, or compiled languages.
- Versions - by publishing a version of your function, you can store your code and configuration as separate stable resources

Invoke/Test

- Synchronous invocation
- Testing locally and in the cloud with the help of AWS SAM templates and AWS SAM CLI

Update / Package

- Updating code and dependencies
- Packaging with the help of AWS SAM templates and AWS SAM CLI

Monitor

- Logs in CloudWatch
- Errors and tracing in X-Ray
- Metrics

Secure

- Execution role

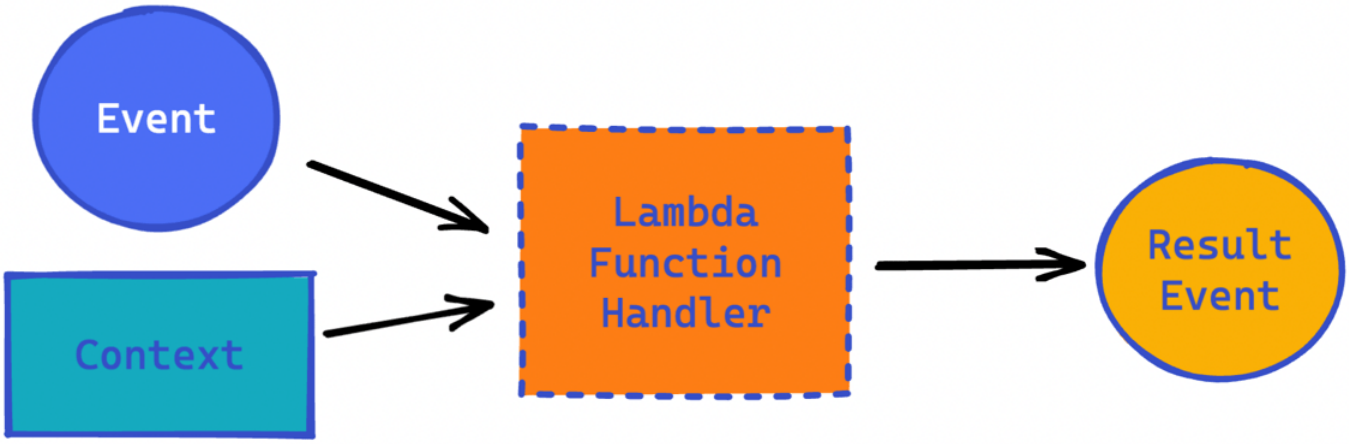
- Policies that grant least privilege to your functions

Workshop - Intro to Serverless - Before diving too deep, you can choose to try out serverless in a workshop or tutorial. Connect to a data source and create a REST API with your first Lambda function."

- Services used: AWS Management Console, Lambda, DynamoDB, API Gateway

Programming Model

The Lambda service provides the same event-based programming model for all languages. The Lambda runtime passes an *invocation event* and *context* to your Lambda function *handler* which does some work and produces a resulting event:



The *invocation event* contains data, as a JSON packet, which varies from service to service. For example, API gateway events include path, HTTP method, query string parameters, headers, cookies, and more. DynamoDB events could contain updated or delete record data. S3 events include the bucket name and object key, among other things.

The context contains information about the environment the function is running inside. Additional contextual information can be set in familiar environment variables (ENV).

The function *handler* is a method in your function code that processes the inbound event. The handler, which is a standard function in your language of choice, does some work and emits a *result event*.

After the handler finishes processing the first event, the runtime sends it another, and another. Each instance of your function could process thousands of requests.

Unlike traditional servers, Lambda functions do not run constantly. When a function is triggered by an event, this is called an *invocation*. Lambda functions are limited to 15 minutes in duration, but on average, across all AWS customers, most invocations last for less than a second.

There are many types of invocation events. Some examples:

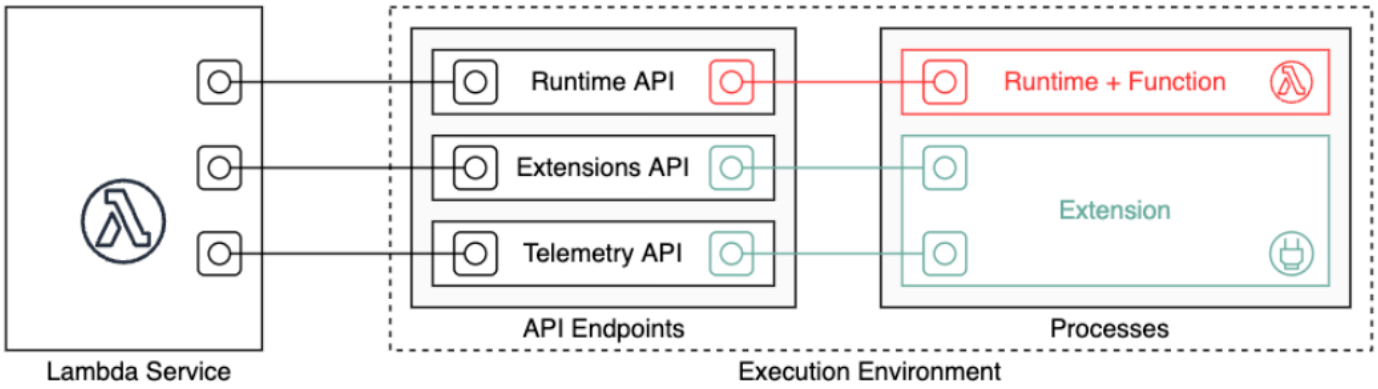
- HTTP request from API Gateway
- Schedule managed by an EventBridge rule
- Message from an IOT device
- Notification that a file was uploaded to an S3 bucket

Even the smallest Lambda-based application uses at least one event that invokes your function.

How Lambda invokes your function (runtime environment)

Lambda invokes your function in an *execution environment*, which contains a secure and isolated *runtime environment*.

- A *runtime* provides a language-specific environment which relays invocation events, context information, and responses between the Lambda and your functions.
- An *execution environment* manages the processes and resources that are required to run the function.



You can use runtimes that Lambda provides for JavaScript (Node.js), TypeScript, Python, Java, Python, Go, C#, and PowerShell, or you can build your own custom runtime environment inside of a container.

If you package your code as a .zip file archive, you must configure your function to use a runtime that matches your programming language. For a container image, you include the runtime when you build the image.

How to process events with a Lambda handler

Conceptually, there are only three steps to processing events with Lambda:

1. Configure the entry point to your function, known as the *handler*, and deploy the function.
2. Lambda service initializes the function, then it invokes the *handler* with an invocation event and context.
3. Your handler function processes the event and returns a response event.

Subsequent events will invoke the handler again, without the initialization delay. During this cycle, the function stays in memory, so clients and variables declared outside of the handler method can be reused.

After a period of time, Lambda will eventually tear down the runtime. This can happen for a variety of reasons; some examples: scaling down to conserve resources, updating the function, updating the runtime.

The function **handler** is the essential component of your function code. As noted previously, the handler is the entry point, but it may not be the only function in your code. In fact, a best practice is keeping the handler sparse and doing the actual processing in other functions in your code.

Here are some example **handlers**:

Python

```
# Example handler method in Python
def lambda_handler(event, context):
  message = 'Hello {} {}!'.format(event['first_name'], event['last_name'])
  return {
       'message' : message
      }
```

Node.js

```
# Example handler method for Node.js
exports.handler =  async function(event, context) {
  console.log("EVENT: \n" + JSON.stringify(event, null, 2))
  return context.logStreamName
  }
```

Java

```
# Example handler method in Java
package example;
import com.amazonaws.services.lambda.runtime.Context
import com.amazonaws.services.lambda.runtime.RequestHandler
import com.amazonaws.services.lambda.runtime.LambdaLogger

// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String,String>, String>{
  Gson gson = new GsonBuilder().setPrettyPrinting().create();

  @Override
  public String handleRequest(Map<String,String> event, Context context)
  {
    LambdaLogger logger = context.getLogger();
    String response = new String("200 OK");
    logger.log("EVENT: " + gson.toJson(event));
    return response;
  }
}
```

C#

```
// Example handler method in C#
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeriali

namespace HelloWorld;

public class Function
```

```
{
    public string FunctionHandler(string input, ILambdaContext context)
    {
        context.Logger.LogLine($"Transforming {input} to upper case");

        return input.ToUpper();
    }
}
```

Handlers in interpreted languages can be deployed directly through the web-based AWS Management Console. Compiled languages, such as Java and C#, or functions that use external libraries are deployed using .zip file archives or container images. Because of that additional process, this guide will focus on Python for examples.

Regardless of language, Lambda functions will generally return a *response event* on successful completion. The following program listing is an example response event to send back to API Gateway so that it can handle a request:

```
{
  "statusCode": 200,
  "headers": {
    "Content-Type": "application/json"
   },
    "isBase64Encoded": false,
    "multiValueHeaders": {
      "X-Custom-Header": ["My value", "My other value"],
    },
  "body": "{\n  \"TotalCodeSize\": 104330022,\n  \"FunctionCount\": 26\n}"
}
```

How to write logs with serverless applications

You might have noticed the logging statements in the preceding handler code. Where do those log messages go?

During invocation, the Lambda runtime automatically captures function output to Amazon CloudWatch.

In addition to logging your function's output, the runtime also logs entries when function invocation starts and ends. This includes a report log with the request ID, billed duration,

initialization duration, and other details. If your function throws an error, the runtime returns that error to the invoker.

To help simplify troubleshooting, the AWS Serverless Application Model CLI (AWS SAM CLI) has a command called sam logs which will show you CloudWatch Logs generated by your Lambda function.

For example, the following terminal command would show the live tail of logs generated by the *YourLambdaFunctionName* Lambda function:

```
sam logs -n YourLambdaFunctionName --tail
```

Logging and debugging go hand in hand. Traces of events are available with Amazon X-Ray for debugging.

Securing functions

AWS Identity and Access Management (IAM) is the service used to manage access to AWS services. Lambda is fully integrated with IAM, allowing you to control precisely what each Lambda function can do within the AWS Cloud. There are two important things that define the scope of permissions in Lambda functions:

- *resource policy*: Defines which events are authorized to invoke the function.
- *execution role policy*: Limits what the Lambda function is authorized to do.

Using IAM roles to describe a Lambda function's permissions, decouples security configuration from the code. This helps reduce the complexity of a lambda function, making it easier to maintain.

A Lambda function's resource and execution policy should be granted the minimum required permissions for the function to perform it's task effectively. This is sometimes referred to as the rule of least privilege. As you develop a Lambda function, you expand the scope of this policy to allow access to other resources as required.

Advanced Topics

You can do a lot by just creating a function and connecting it to an event source like API Gateway or S3 triggers.

As you progress on your journey, you should explore the following more advanced topics.

- Connect services with event source mapping
- Deploy code in containers
- Add additional code with layers
- Augment functions with extensions
- Launch functions faster with SnapStart
- Connect to functions with Function URLs

Event source mapping

Some services can trigger Lambda functions directly, for example, when an image is added to an S3 bucket, a Lambda can be triggered to resize it. Some services cannot invoke Lambda directly; but you can instead use an *event source mapping* which is a polling mechanism that reads from an event source and invokes a Lambda function.

You can use event source mappings to process items from a stream or queue in the following services:

- Amazon DynamoDB
- Amazon Kinesis
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- Self-managed Apache Kafka
- Amazon Simple Queue Service

Related resource:

- Event source mapping official documentation, including the default behavior that batches records together into a single payload that Lambda sends to your function.

Deploy with containers

If you need a custom runtime that is not provided by AWS, you can create and deploy a custom container image. AWS provides base images preloaded with a language runtime and other components that are required to run the image on Lambda. AWS provides a Dockerfile for each of the base images to help with building your container image.

Custom containers are one way you might experiment with lift and shift of existing code to Lambda runtimes. If you do this, consider the architectural differences between always running containers, versus on demand nature of Lambda functions.

Related resource:

- Deploy container images

Add code with Layers

A Lambda *layer* is a .zip file archive that can contain additional code or other content. A layer can contain libraries, a custom runtime, data, or configuration files. Layers are also necessary if your function .zip archive exceeds the size limit.

Layers provide a convenient way to package libraries and other dependencies that you can use with your Lambda functions. Using layers reduces the size of uploaded deployment archives and makes it faster to deploy your code. Layers also promote code sharing and separation of responsibilities so that you can iterate faster on writing business logic.

Related resource:

- Creating and sharing Lambda layers

Extensions

You can use Lambda extensions to augment your Lambda functions. For example, use Lambda Extensions to integrate with your preferred monitoring, observability, security, and governance tools.

Lambda supports internal or external extensions. An internal extension runs as part of the runtime process. An external extension runs as an independent process in the execution environment and continues to run after the function invocation is fully processed.

Related resources:

- Datadog Lambda Extension - an extension that supports submitting custom metrics, traces, and logs asynchronously while your Lambda function executes.
- Lambda Extensions - official documentation

Launch functions faster with SnapStart

Lambda SnapStart for Java can improve startup performance by up to 10x at no extra cost, typically with no changes to your function code. The largest contributor to startup latency (often referred to as cold start time) is the time that Lambda spends initializing the function, which includes loading the function's code, starting the runtime, and initializing the function code.

With SnapStart, Lambda initializes your function when you publish a function version. Lambda takes a Firecracker microVM snapshot of the memory and disk state of the initialized execution environment, encrypts the snapshot, and caches it for low-latency access.

Note: You can use SnapStart only on published function versions and aliases that point to versions. You can't use SnapStart on a function's unpublished version ($LATEST).

Related resources:

- Accelerate Your Lambda Functions with Lambda SnapStart - an AWS Compute blog article by Jeff Barr from Nov 2022 that shows the configuration change and vast difference from roughly six seconds init time to 142 milliseconds of restore time with SnapStart

Connect to functions with Function URLs

A function URL is a dedicated HTTP(S) endpoint for your Lambda function. You can create and configure a function URL through the Lambda console or the Lambda API. When you create a function URL, Lambda automatically generates a unique URL endpoint for you. Once you create a function URL, its URL endpoint never changes. Function URL endpoints have the following format:

```
https://<url-id>.lambda-url.<region>.on.aws
```

After you configure a function URL for your function, you can invoke your function through its HTTP(S) endpoint with a web browser, curl, Postman, or any HTTP client.

Related resources:

- Function URLs - official documentation

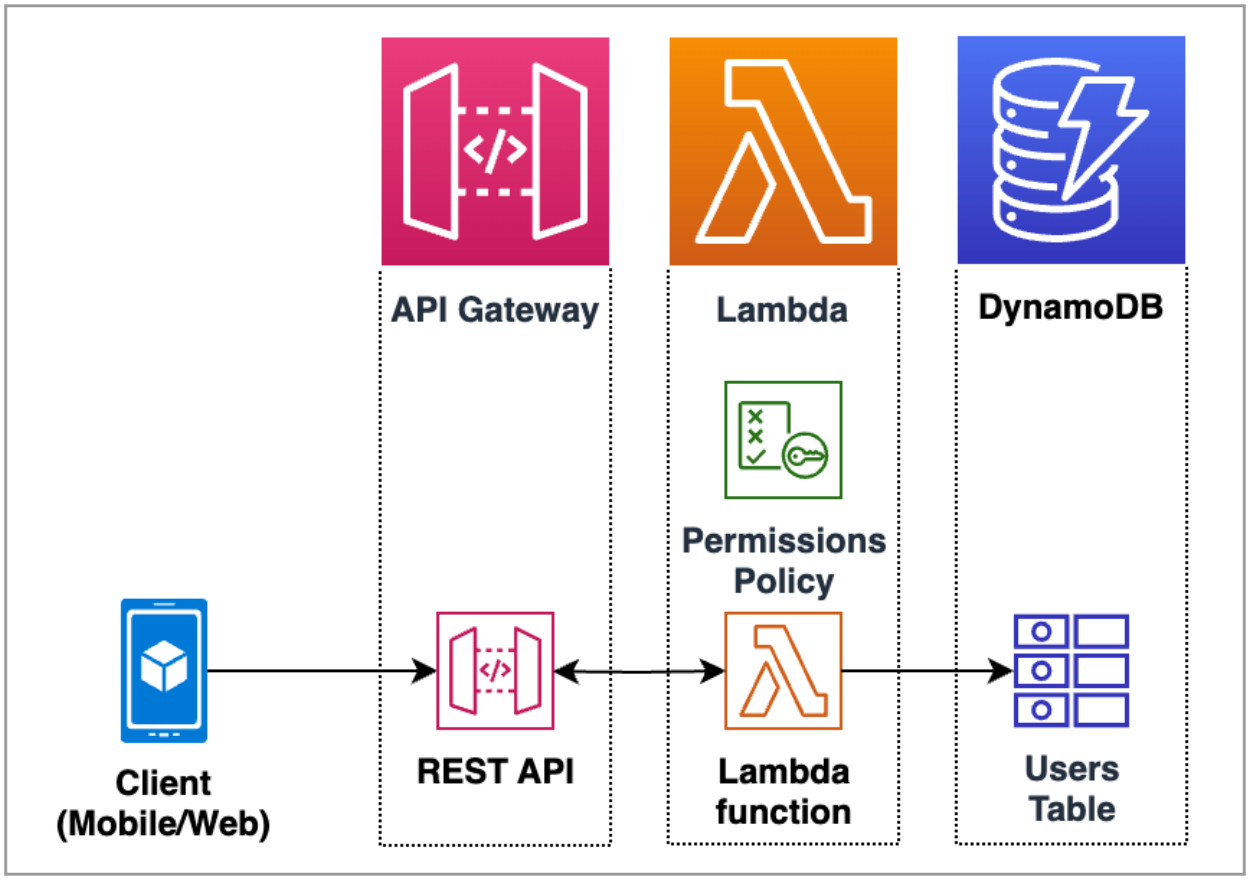
Additional resources

Official AWS documentation:

- AWS Lambda Developer Guide - extensive and complete documentation for Lambda

Next steps

> **Learn serverless techniques in an online workshop**
>
> Learn by doing in the **Serverless Patterns Workshop**. The first module introduces a serverless microservice to retrieve data from DynamoDB with Lambda and API Gateway. Additional modules provide practical examples of unit and integration testing, using infrastructure as code to deploy resources, and how to build common architectural patterns used in serverless solutions.



Get started with API Gateway

Desktop and mobile browsers, command line clients, and applications all make requests to your web-based APIs. Your application API must handle these requests, scale based on incoming traffic, ensure secure access, and be available in multiple environments.

For serverless applications, API Gateway acts as the entry point, also called the “front door”, for your web-based applications.

You are likely familiar with how to use and setup back-end APIs in traditional application frameworks. In this starter, you will learn the essential role Amazon API Gateway plays in the request/response workflow. Additionally, API Gateway can optimize response time with cache, prevent function throttling, and defend against common network and transport layer DDoS attacks.

What is API Gateway?

Amazon API Gateway is an AWS service for creating, publishing, maintaining, monitoring, and securing REST, HTTP, and WebSocket APIs. API Gateway can transform inbound web requests into *events* that are processed by Lambda functions, AWS services, and HTTP endpoints.

API Gateway can process many concurrent API calls and provides additional management capabilities, including:

- Authorization, authentication, and API Keys for access control
- Documentation and version control
- OpenAPI 3.0 import and export

Although similar in purpose, the REST API and HTTP API configuration and management console experiences are quite different. WebSocket APIs are included in the advanced section, as they are less frequently used when starting out.

- REST API configuration models **resources** with Client, Method request, Integration Request, Integration endpoint, Integration Response, Method Response, return to Client.
- HTTP API configuration models **routes** with a tree of HTTP methods and associated integration configuration.

> **Important**
>
> Unless otherwise stated, REST APIs are discussed in the fundamentals.

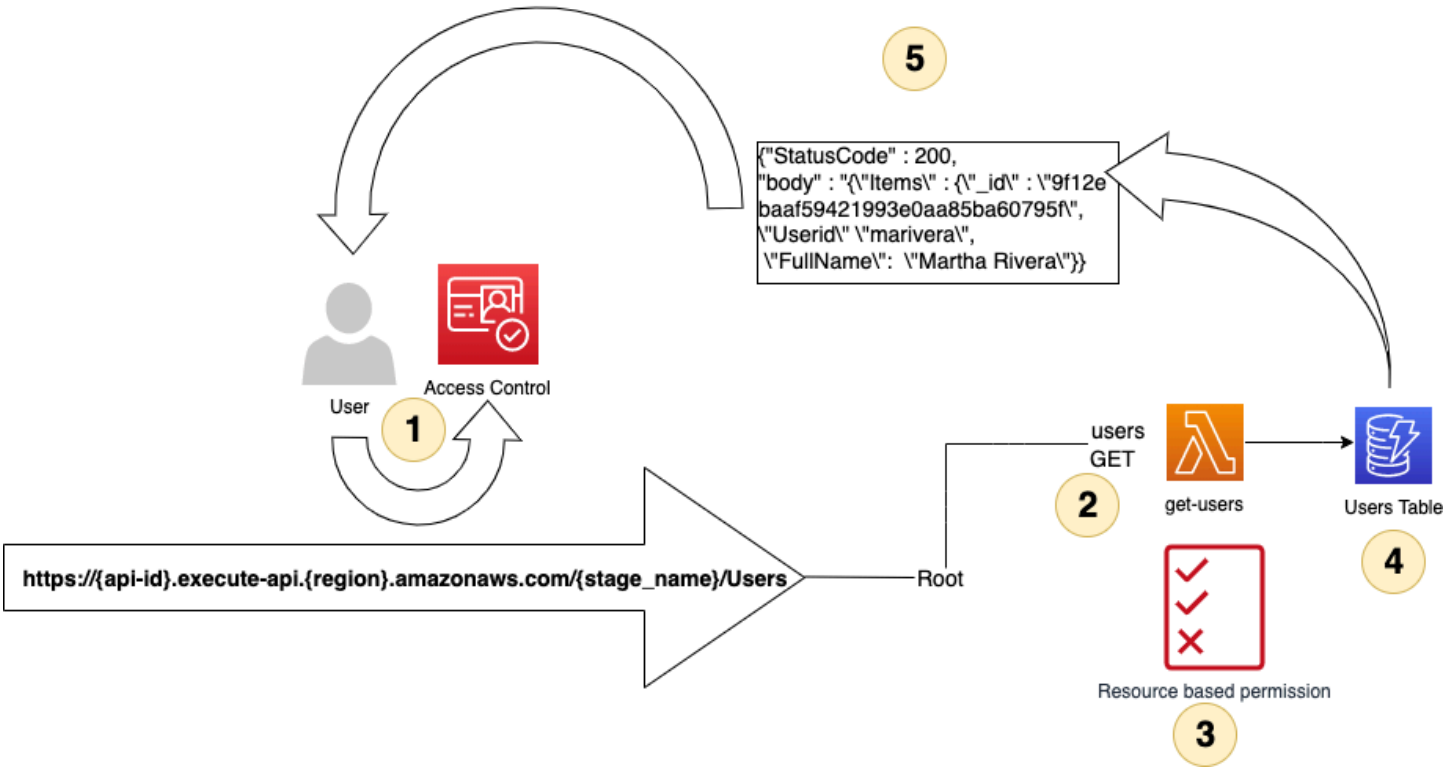
Fundamentals

An API (Application Program Interface) is a collection of endpoints that connect to your application logic. You may already be familiar with creating APIs in traditional frameworks. This section will introduce how the concepts map to API Gateway.

Core Concepts

You can choose to get started with the core concepts in a workshop or tutorial, or read through the high level concepts here.

Let's start with a high level work flow for an API request, pictured in the following diagram. Imagine a UI component that requests User data from the server to show a table of users:



- API HTTP request for a user is received and authentication is checked
- The call matched the API `GET` method and `users` resource, which is integrated with a get-users Lambda function
- Permissions were verified before invoking the AWS resource
- Lambda function sends queries to retrieve items from a data store
- Data is wrapped in an HTTP response to be returned to the client

Create an API

When you create an API Gateway REST API, there are two essential components:

- **method** – HTTP methods (`GET`, `POST`, `PUT`, `PATCH`, `OPTIONS` and `DELETE`) An HTTP method is an action to be taken on a resource. API Gateway also has an `ANY` method, that matches any method type.
- **resource** – A resource is related to your business logic, for example users, orders, or messages.

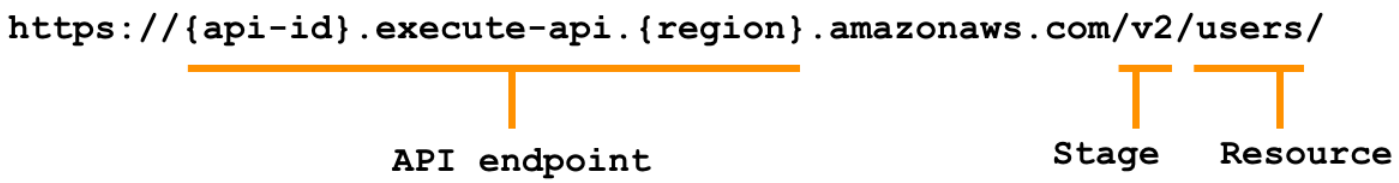
Resources can also have the following:

- **child resources** - such as `/users/PremiumUser` to retrieve special types of Users
- **path parameters** – such as `/users/{UserId}` to retrieve a specific user by an identifier like 12345 or UserA

Resources must have an *integration* to connect the resource to a back-end endpoint:

- **integration** – connection to a Lambda function, HTTP endpoint, or AWS service action

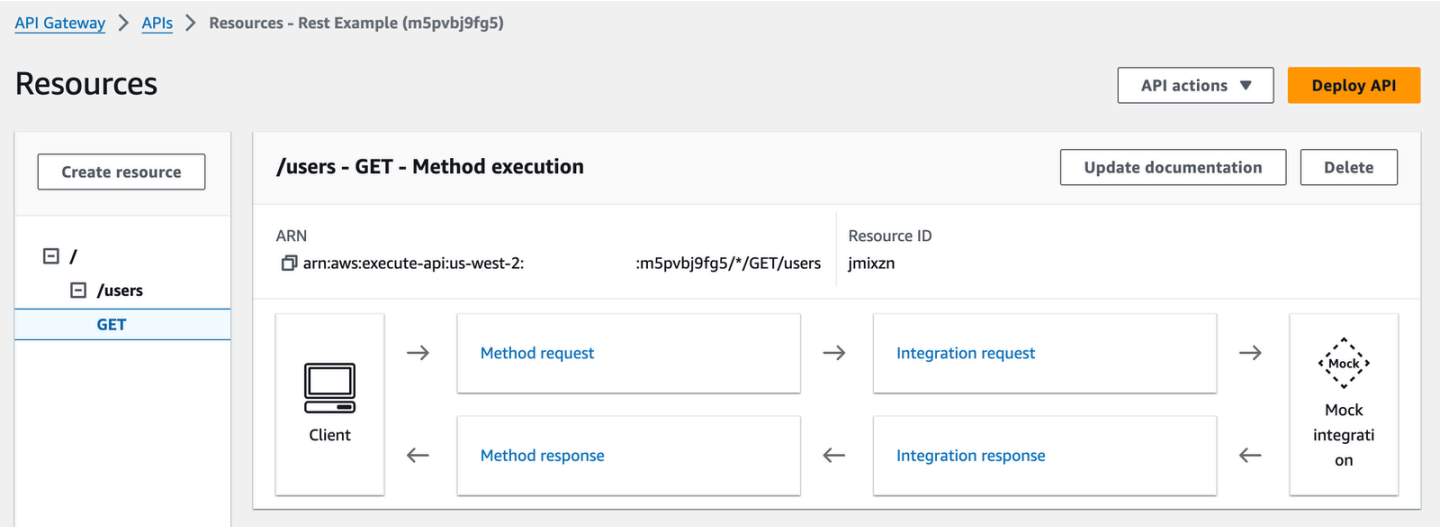
The following diagram shows the components of a URL request: API Endpoint, Stage, and Resource.



- **API Endpoint**: The hostname for the API. Unless you designate a custom domain name, all APIs created using API Gateway will have this structure.
- **Stage Name:** An API deployment. You can deploy different snapshots of your API to various stages, for example: “v2", "latest", "dev", "qa".
- **Resource**: The piece of your business logic provided by the request.

To create an REST API resource, you specify the resource path, then add a method with an API integration endpoint. You can then further configure the integration and test it in the console. The following screenshot shows an example REST API integration for a GET method for the /users resource.

After you are satisfied with your configuration, you must **deploy** the API to a **stage** so it will become available to process requests.



Integrate your API resources

During the creation of an API method, you must integrate it with an backend endpoint. A backend endpoint is also referred to as an *integration endpoint* and can be one of the following:

- Lambda function to invoke
- HTTP server to forward the request to
- AWS service action to invoke

Integration of your API is how your frontend and backend communicate.

Like an API request, an API integration has an *integration request* and an *integration response*. The integration request encapsulates the HTTP request received by the backend. The integration response is an HTTP response that wraps the output returned by the backend.

Setting up an integration request involves the following:

- Configuring how to pass client-submitted method requests to the backend
- Configuring how to transform the request data (REST API only), if necessary, to the integration request data
- Specifying an integration endpoint: Lambda function, HTTP server, or AWS service action

For advanced non-proxy integrations (REST API only), setting up an integration response involves the following:

- Configuring how to pass the backend-returned result to a method response with a given HTTP status code
- Configuring how to transform specified integration response parameters to preconfigured method response parameters
- Configuring how to map the integration response body to the method response body according to the specified body-mapping templates.

In the simpler case of proxy integrations, the preceding steps are handled automatically.

Invoke your API

After you have deployed your REST API, you can invoke it. The following shows the standard format of an API URL

```
https://{restapi_id}.execute-api.{region}.amazonaws.com/{stage_name}/
```

- `{restapi_id}` is your unique API identifier
- `{region}` is the AWS region, or location of the API
- `{stage_name}` is the stage name of the API deployment.

You can call an API using a web browser, through the CLI using cURL or the Postman application. After you deploy your API, you can turn on invocation logs using CloudWatch to monitor your API calls.

You can call a REST API **before** deployment for testing in two ways:

- In the API Gateway console by using the API Gateway's `TestInvoke` feature.
- Through the CLI using the `test-invoke-method` command.

Both of these methods bypass the `Invoke` URL and any authorization steps to allow API testing.

Alternatively, after the API is successfully deployed, you can use any command line or graphical tool, like cURL or Postman to call your API.

Related resource(s):

- cURL home page - Learn how to invoke APIs with cURL

- Postman home page - Learn how to invoke APIs with the Postman application

Protect your API

To authenticate and authorize access to your Rest APIs, you can choose from the following:

- Amazon Cognito user pools as an identity source for who access the API.
- Lambda functions to control access to APIs by using a variety of identity sources.
- Resource-based policies to allow or deny specified access from source IP addresses or VPC endpoints.
- AWS Identity and Access Management roles, policies, and IAM tags to control access for who can invoke certain APIs.

Related resource(s):

- Controlling and managing access to a REST API in API Gateway
- API Security Whitepaper - Learn about AWS best practices for API Gateway security

More resources related to access control:

- How to set up cross-origin resource sharing (CORS)
- Mutual TLS in REST API
- Control access with AWS WAF

Advanced Topics

You can connect a microservice with a REST API and a single integration to a Lambda function. As you progress on your journey, you should explore the following advanced topics.

- Choose between REST and HTTP APIs
- Explore non-proxy custom integrations and data transformation
- Optimize your API with caching
- Explore HTTP APIs
- Learn how to use proxy routes and resources
- Send binary media types
- Explore bidirectional connections using WebSocket APIs
- Build and export your API using an OpenAPI 3 definition

Choose between REST and HTTP APIs

API Gateway offers two types of RESTful API products: REST APIs and HTTP APIs .

- REST APIs support more features than HTTP APIs, but pricing is higher than HTTP APIs.

- HTTP APIs consist of a collection of *routes* that direct incoming API request to backend resources. Routes are a combination of an HTTP method and a resource path, such as “GET /users/details/1234”. HTTP APIs also provide a `$default` route that is a catch-all for requests that don’t match other more specific routes.

Choosing between the type of API depends on your specific use case:

- Choose a REST API if you need advanced features, such as mock integration, request validation, a web application firewall, certificates for backend authentication, or a *private* API endpoint with per-client rate limiting and usage throttling,
- Choose an HTTP API if you need minimal features, lower price, and auto-deployment.

Related resource(s):

- Choosing between REST APIs and HTTP APIs - detailed comparison between REST and HTTP APIs

Non-proxy integrations and data transformations (REST API only)

Your API integration contains an integration request and integration response. You can have API Gateway directly pass the request and response between the frontend and backend, or you can manually set up an integration request and integration response.

- *proxy integration* – you let API Gateway automatically pass all data in the HTTP request/response between the client and backend, automatically, without modification
- *non-proxy integration* – you set up a custom integration request and integration response, where you can alter the data that flows between client and backend

Choosing between integration types depends on your use case:

- **Proxy integrations** directly send all information to a function for processing.

- **Non-proxy custom integrations** can transform data before it gets to your integration service and before the output is sent to clients. For Lambda, your function code can focus on the business task rather than parsing data in the input event. Non-proxy can be a good fit for legacy code migrations too, because you can transform the data to match expectations of existing code.

Related resource(s):

- Tutorial: Build a Hello World REST API with Lambda proxy integration (proxy example)
- Using API Gateway as a proxy for DynamoDB (custom non-proxy integration example)
- Best practices for working with Apache Velocity Template Language (VTL) (custom non-proxy integration example)

Optimize your API with caching (REST API only)

To reduce the number of calls made to your endpoint and improve the latency of requests to your API, you can cache responds from your endpoint for a specified time-to-live period. API Gateway will respond to the request by using the cache instead of your end point. This can speed up your latency.

Related resource(s):

- Enabling API caching to enhance responsiveness - Learn how to enable caching for your REST APIs.

Send binary media types

You can use either REST APIs or HTTP APIs to send binary payloads such as a JPEG or gzip file. For REST APIs, you need to take additional steps to handle binary payloads for non-proxy integrations.

Related resource(s):

- Return binary media from a Lambda proxy integration - Learn how to use a Lambda function to return binary media. This works for both REST and HTTP APIs.
- Working with binary media types for REST APIs - Additional considerations for REST non-proxy integrations

Greedy path variables

If you want to handle the routing implementation outside of API Gateway, for example inside a Lambda function, you can use a greedy path variable in the form of a proxy resource `{proxy+}` to match all the child resources of a route for both HTTP and REST APIs.

Here are two examples of using greedy path variables:

- Use `/{proxy+}` to match both `/users` and `/users/{UserID}` routes
- Use the `ANY` method for a `/{proxy+}` resource at the root of your API to match all HTTP method types

Related resource(s):

- Use a proxy resource to streamline API setup – Set up a proxy resource and greedy path variable for REST APIs.
- Working with routes for HTTP APIs – Set up a greedy path variable for HTTP APIs

WebSocket APIs

WebSocket APIs are a connection of WebSocket routes that are integrated with backend HTTP endpoints, Lambda functions, or other AWS services. A client can send messages to a service, and services can independently send messages to clients, making these APIs bidirectional. Because of this, WebSocket APIs are often used as chat applications or for multiplayer games or financial trading platforms.

Related resource(s):

- About WebSocket APIs in API Gateway - Get started with WebSocket APIs
- Tutorial: Building a serverless chat app with a WebSocket API, Lambda, and DynamoDB - Intermediate level WebSocket API tutorial using AWS CloudFormation

OpenAPI

The OpenAPI 3 definition allows you to import and export APIs using API Gateway. OpenAPI is a standard format that is language agnostic and vendor-neutral and is used to define and structure REST APIs. There are many Open API extensions to support the AWS-specific authorization and API Gateway-specific API interactions for REST APIs and HTTP APIs.

You can use OpenAPI API definitions in AWS SAM templates for more complicated applications. Or, you can build APIs with API Gateway and export the OpenAPI 3.0 definition to use with other services.

Related resource(s):

- OpenAPI FAQ - Introduction and FAQ for the OpenAPI Specification
- Working with API Gateway extensions to OpenAPI - Learn how to use API Gateway extensions to the OpenAPI specification

Additional resources

Official AWS documentation:

- API Gateway Developer Guide - extensive and complete documentation for Amazon API Gateway
- REST API tutorial - REST API tutorial
- HTTP API tutorial - HTTP API tutorial

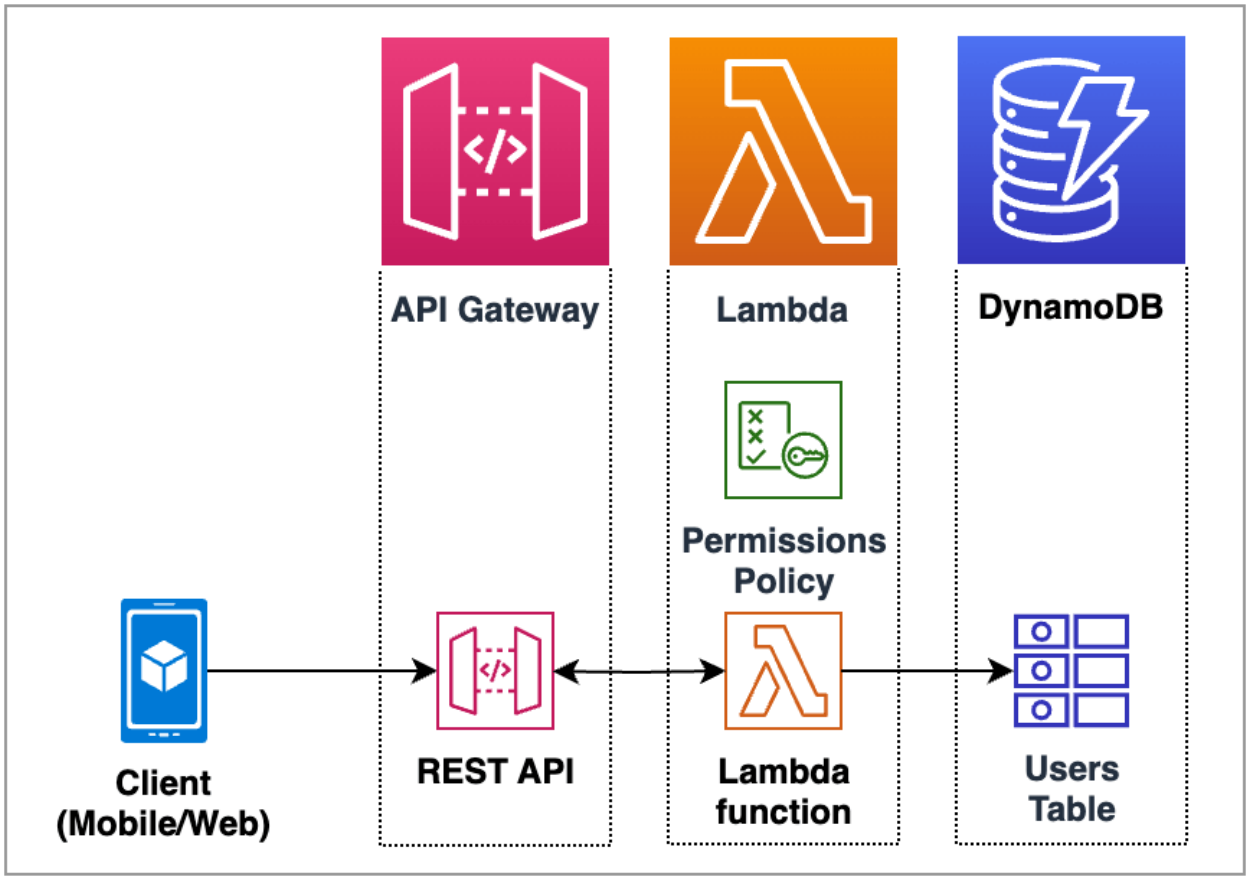
Resources from the serverless community:

- Building Happy Little APIs | I Didn't Know Amazon API Gateway Did That - AWS video series introducing to API Gateway

Next Steps

Learn how to use API Gateway in an online workshop

Learn by doing in the **Serverless Patterns Workshop**. The first module introduces a serverless microservice to retrieve data from DynamoDB with Lambda and API Gateway. Additional modules provide practical examples using infrastructure as code to deploy resources, test, and build with common architectural patterns used in serverless solutions.



Get started with DynamoDB

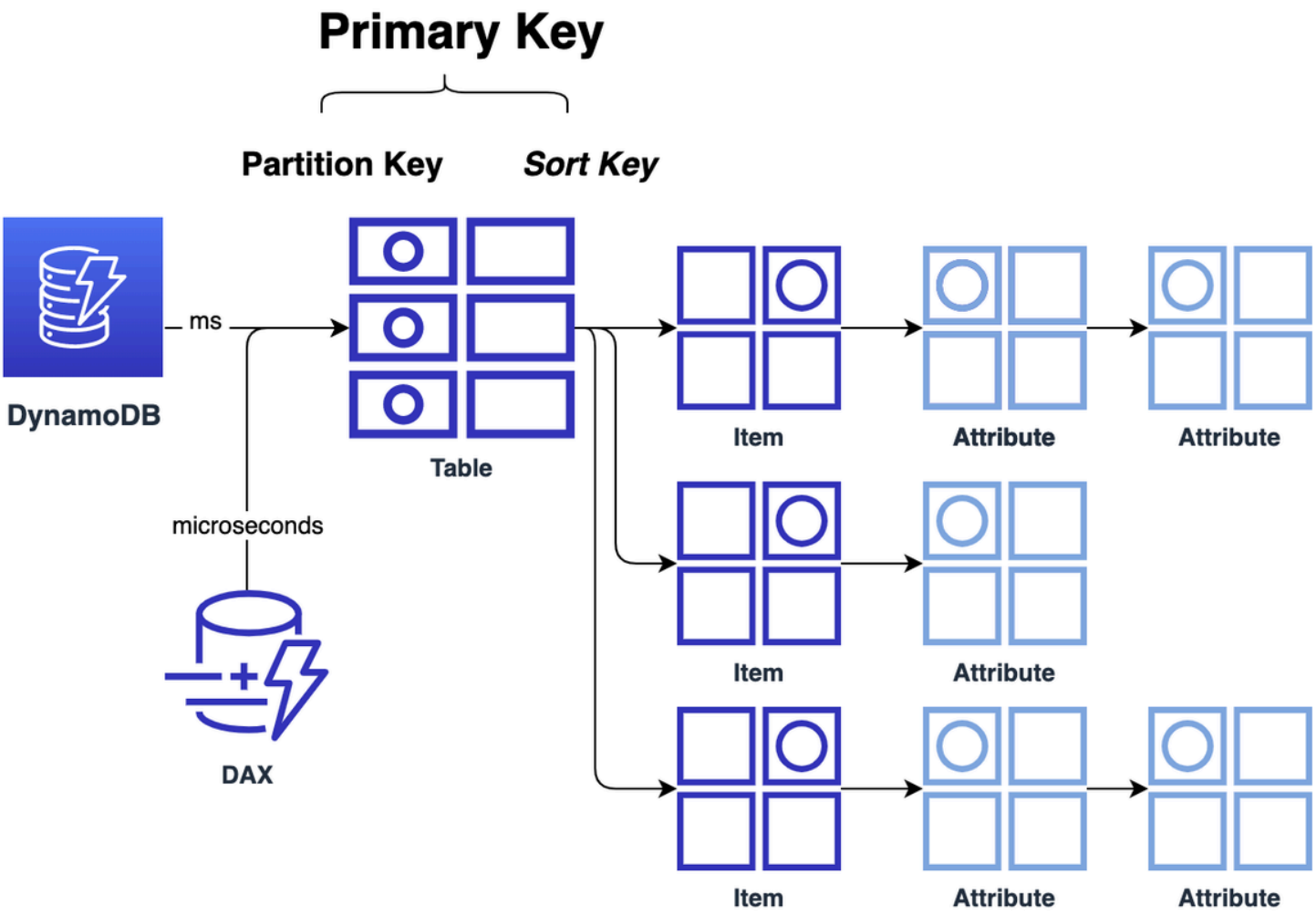
Application logic is important, but an essential component is **your data**.

You are likely familiar with storing data in SQL and NoSQL databases in traditional solutions. Due to its rapid response and low latency, Amazon DynamoDB, a NoSQL data store released in 2012 is a frequently used data storage service for serverless solutions.

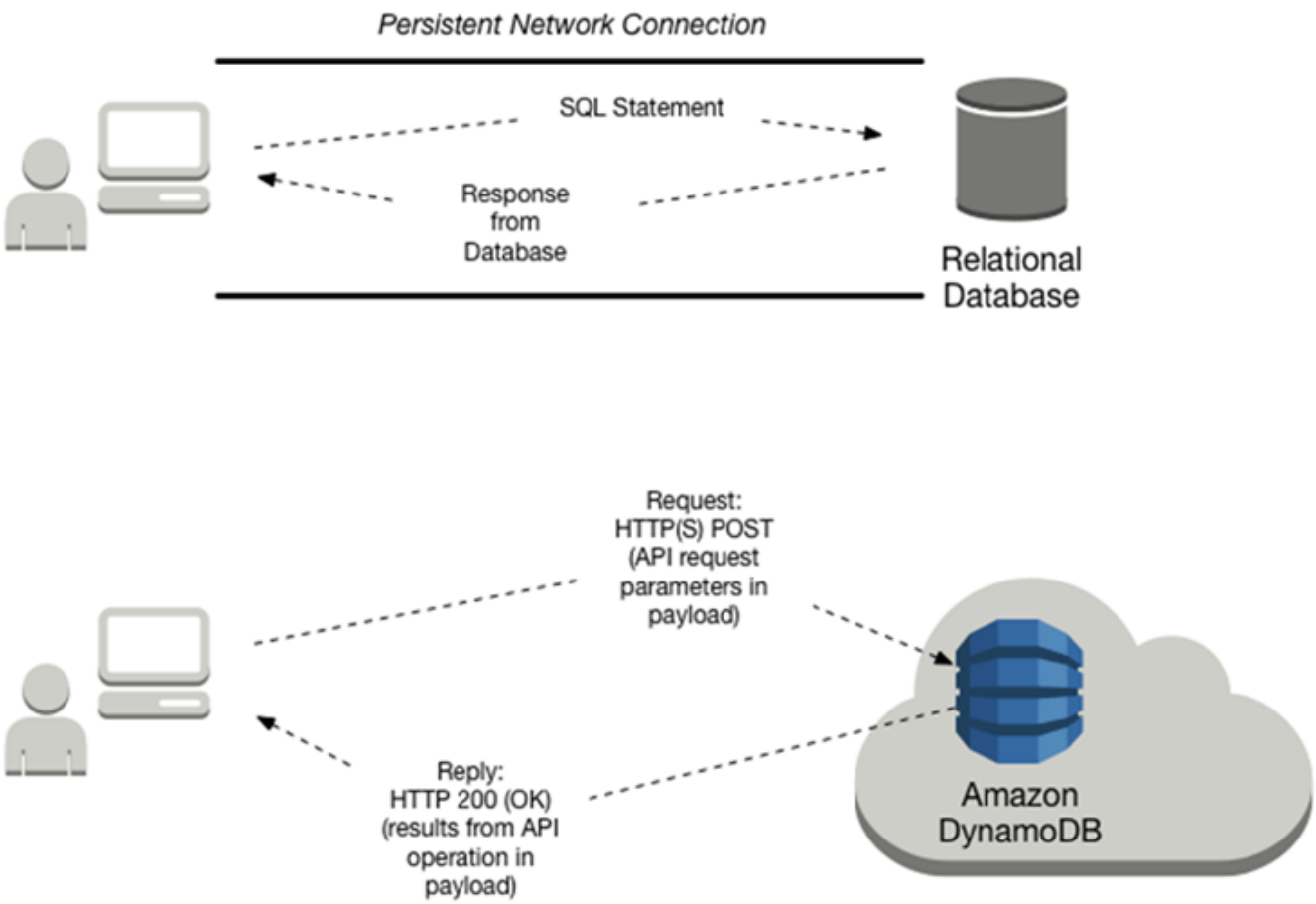
What is DynamoDB?

Amazon DynamoDB is a fully managed serverless NoSQL database service. DynamoDB stores data in *tables.* Tables hold *items.* Items are composed of *attributes*. Although these components sound similar to a traditional SQL table with rows and fields, there are also differences which will be explained in the fundamentals section.

Data access is generally predictable and fast, in the millisecond (ms) range. If you need even faster response time, the DynamoDB Accelerator (DAX) provides in-memory acceleration for microsecond level access to data.



Traditional web frameworks maintain persistent network connections to SQL databases with *connection pools* to avoid latency accessing data. With serverless architecture and DynamoDB, connection pools are **not** necessary to rapidly connect and scale the database. Instead, you can adjust your tables' throughput capacity, as needed.



For rapid local development, modeling, and testing, AWS provides a downloadable version of DynamoDB that you can run on your computer. The local database instance provides the same API as the cloud-based service.

Fundamentals

In DynamoDB, *tables*, *items*, and *attributes* are the core components. Data items stored in tables are identified with a *primary key*, which can be a simple *partition hash key* or a composite of a partition key and a sort key. Although these terms may sound familiar, we will define all of them to clarify how similar and different they are from traditional SQL database terms.

Core Components

A *table* is a collection of *items*, and each *item* is a collection of *attributes*.

- **Table** – a collection of data. For example, a table called *People* could store personal contact information about friends, family, or anyone else of interest. You could also have a *Cars* table to store information about vehicles that people drive.

Data in a DynamoDB is uniquely identified with a *primary key,* and optional secondary indexes for query flexibility. DynamoDB tables are schemaless. Other than the *primary key*, you do not need to define additional attributes when you create a table.

Each table contains zero or more *items.*

- **Item** – An *item* is a group of attributes that is uniquely identifiable among all of the other items. In a *People* table, each item represents a person. For a *Cars* table, each item represents one vehicle.

Items in DynamoDB are similar to rows, records, or tuples in other database systems. In DynamoDB, there is no limit to the number of items you can store in a table. DynamoDB items have a size limit of 400KB. An *item collection*, a group of related items that share the same partition key value, are used to model one-to-many relationships. (1)

Each item is composed of one or more attributes:

- **Attribute** –An *attribute* is a fundamental data element, something that does not need to be broken down any further. For example, an item in a *People* table contains attributes called *PersonID*, *LastName*, *FirstName*, and so on. In a Cars table, attributes could include *Make, Model, BuildYear, and RetailPrice.* For a *Department* table, an item might have attributes such as *DepartmentID*, *Name*, *Manager*, and so on. Attributes in DynamoDB are similar in many ways to fields or columns in other database systems.

Most of attributes are *scalar*, which means that they can have only one value. Strings and numbers are common examples of scalars. Attributes may be nested, up to 32 levels deep. An example could be an Address which contains Street, City, and PostalCode.

Watch an AWS Developer Advocate explain these core concepts in this video: Tables, items, and attributes (6 min).

As mentioned in the video, the primary key for the following table consists of both a partition key and sort key. The sort keys “inventory::armor” and “inventory::weapons” contain double colons to

add query flexibility to get all inventory. This is not a DynamoDB requirements, just a convention by the developer to make retrieval more flexible.

Primary key		Attributes		
Partition key: PK	Sort key: SK			
account1234	inventory::armor	data {"armor": [{"name" : "Pauldron of the Paladin", "type" : "chest", "gear score" : 545 }, { "name" : "Greaves of the Ranger", "type" : "sword", "gear score" : 382}]}		
	inventory::weapons	data {"weapons" : [{ "name" : "Sword of the Ancients", "type" : "sword", "gear score" : 320}]}		
	login-data	pw d1e8a70b5ccab1dc2f56bbf7e99f064a660c08e361a35751b9c483c88943d082	state Active	last-login 1649276737

All of the data for account1234 will be stored in the same database partition to ensure retrieval of related data is quick.

Related resources:

- Item collections - how to model one-to-many relationships in DynamoDB - example of using an item collection, a group of related items that share the same partition key value, as a way to model one-to-many relationships

- Characteristics of databases - comparison of SQL and NoSQL qualities of DynamoDB

Reading data

DynamoDB is a non-relational NoSQL database that does not support table joins. Instead, applications read data from one table at a time. There are four ways to read data:

- `GetItem` – Retrieves a single item from a table. This is the most efficient way to read a single item because it provides direct access to the physical location of the item. (DynamoDB also provides the `BatchGetItem` operation, allowing you to perform up to 100 `GetItem` calls in a single operation.)

- `Query` – Retrieves all of the items that have a specific partition key. Within those items, you can apply a condition to the sort key and retrieve only a subset of the data. `Query` provides quick, efficient access to the partitions where the data is stored.
- `Scan` – Retrieves all of the items in the specified table. This operation should not be used with large tables because it can consume large amounts of system resources. Think of it like a "SELECT * FROM BIG_TABLE" in SQL. You should generally prefer Query over Scan.
- `ExecuteStatement` retrieves a single or multiple items from a table. `BatchExecuteStatement` retrieves multiple items from different tables in a single operation. Both of these operations use PartiQL, a SQL-compatible query language.

Primary keys and indexes

- **Partition key** - also called a hash key, identifies the partition where the data is stored in the database.
- **Sort key** - also called a range key, represents 1:many relationships

The primary key can be a partition key, nothing more. Or, it can be a composite key which is a combination of a partition key and sort key. When querying, you must give the partition key, and optionally provide the sort key.

Amazon DynamoDB provides fast access to items in a table by specifying primary key values. However, many applications might benefit from having one or more secondary (or alternate) keys available, to allow efficient access to data with attributes other than the primary key. To address this, you can create one or more secondary indexes on a table and issue `Query` or `Scan` requests against these indexes.

A *secondary index* is a data structure that contains a subset of attributes from a table, along with an alternate key to support `Query` operations. You can retrieve data from the index using a `Query`, in much the same way as you use `Query` with a table. A table can have multiple secondary indexes, which give your applications access to many different query patterns.

DynamoDB supports two types of secondary indexes:

- **Global secondary index —** An index with a partition key and a sort key that can be different from those on the base table. A global secondary index is considered "global" because queries on the index can span all of the data in the base table, across all partitions. A global secondary index is stored in its own partition space away from the base table and scales separately from the base table.

- **Local secondary index** — An index that has the same partition key as the base table, but a different sort key. A local secondary index is "local" in the sense that every partition of a local secondary index is scoped to a base table partition that has the same partition key value.

In DynamoDB, you perform `Query` and `Scan` operations directly on the index, in the same way that you would on a table.

Data types

DynamoDB supports many different data types for attributes within a table. They can be categorized as follows:

- **Scalar Types** – A scalar type can represent exactly one value. The scalar types are number, string, binary, Boolean, and null.
- **Document Types** – A document type can represent a complex structure with nested attributes, such as you would find in a JSON document. The document types are list and map.
- **Set Types** – A set type can represent multiple scalar values. The set types are string set, number set, and binary set.

Related resource:

- Supported data types and naming rules in Amazon DynamoDB

Operations on tables

Operations are divided into Control plane, Data plane, Streams, and Transactions:

- *Control plane* operations let you create and manage DynamoDB tables. They also let you work with indexes, streams, and other objects that are dependent on tables. Operations include CreateTable, DescribeTable, ListTables, UpdateTable, DeleteTable.
- *Data plane* operations let you perform create, read, update, and delete (also called *CRUD*) actions on data in a table. Some of the data plane operations also let you read data from a secondary index. Operations include: ExecuteStatement, BatchExecuteStatement, PutItem, BatchWriteItem (to create or delete data), Get Item, BatchGetItem, Query, Scan, UpdateItem, DeleteItem
- *DynamoDB Streams* operations let you enable or disable a stream on a table, and allow access to the data modification records contained in a stream. Operations include: ListStreams, DescribeStreams, GetSharedIterator, GetRecords

- *Transactions* provide atomicity, consistency, isolation, and durability (ACID) enabling you to maintain data correctness in your applications more easily. Operations include: ExecuteTransaction, TransactWriteItems, TransactGetItems

Note: you can also use PartiQL - a SQL-compatible query language for Amazon DynamoDB, to perform data plane and transactional operations.

Advanced Topics

You can do a lot just creating a DynamoDB table with a primary key. As you progress on your journey, you should explore the following more advanced topics.

- Create more complex data models in NoSQL WorkBench.
- Use DynamoDB Streams to trigger functions when data is created, updated, or deleted.
- Coordinate all-or-nothing changes with transactions.
- Query and control the database using SQL-compatible PartiQL query language.
- Reduce millisecond access times to microseconds with the in-memory DynamoDB Accelerator (DAX).

NoSQL Workbench & Local DynamoDB

NoSQL Workbench is a cross-platform visual application that provides data modeling, data visualization, and query development features to help you design, create, query, and manage DynamoDB tables.

- **Data modeling** - build new data models, or design models based on existing data models.
- **Data visualization** - map queries and visualize the access patterns (facets) of the application without writing code. Every facet corresponds to a different access pattern in DynamoDB. You can manually add data to your data model.
- **Operation builder** - use the *operation builder* to develop and test queries, and query live datasets. You can also build and perform data plane operations, including creating projection and condition expressions, and generating sample code in multiple languages.

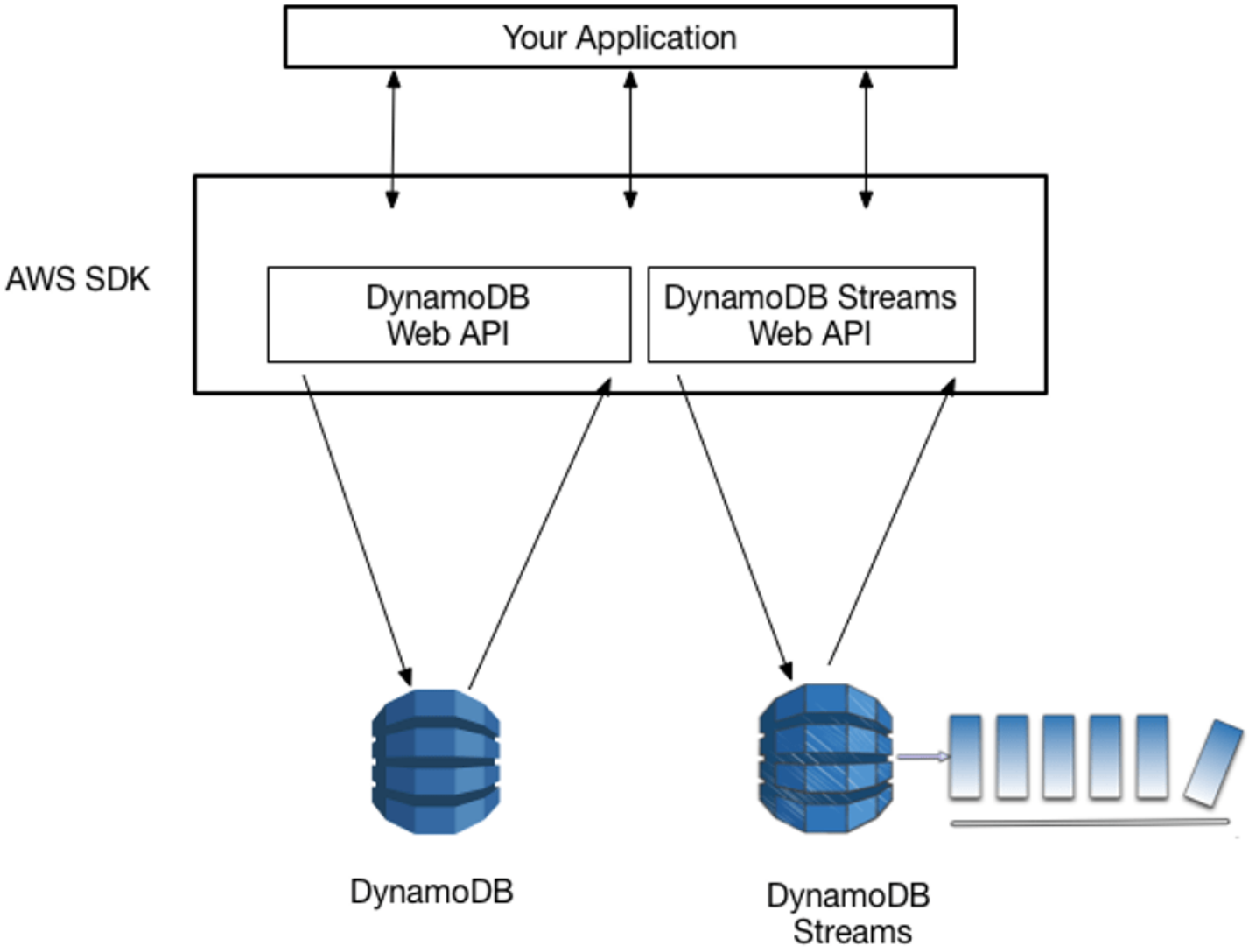
You can also run a local instance of DynamoDB on your workstation. Combined with NoSQL workbench, this can provide a fast local setup for experimentation and learning.

Related resources:

- NoSQL Workbench & Building data models with NoSQL Workbench - model and query data with a desktop tool
- Setting up DynamodDB local (downloadable version)

DynamoDB Streams

DynamoDB Streams is an optional feature that captures data modification events. The data about these events appear in the stream in near-real time, and in the order that the events occurred, as a *stream record*.



If you enable a stream on a table, DynamoDB Streams writes a stream record whenever one of the following events occurs:

- A new item is added to the table: the stream captures an image of the entire item, including all of its attributes.
- An item is updated: the stream captures the "before" and "after" image of any attributes that were modified in the item.

- An item is deleted from the table: the stream captures an image of the entire item before it was deleted.

Each stream record also contains the name of the table, the event timestamp, and other metadata. Stream records have a lifetime of 24 hours; after that, they are automatically removed from the stream.

You can use DynamoDB Streams together with AWS Lambda to create an event source mapping—a resource that invokes your Lambda function automatically whenever an event of interest appears in a stream

For example, consider a *Customers* table that contains customer information for a company. Suppose that you want to send a "welcome" email to each new customer. You could enable a stream on that table, and then associate the stream with a Lambda function. The Lambda function would run whenever a new stream record appears, but only process new items added to the *Customers* table. For any item that has an `EmailAddress` attribute, the Lambda function would invoke Amazon Simple Email Service (Amazon SES) to send an email to that address.

Related resources:

- Change data capture with Amazon DynamoDB -- stream item-level change data in near-real time
- Change data capture for DynamoDB Streams - DynamoDB Streams captures a time-ordered sequence of item-level modifications in any DynamoDB table and stores this information in a log for up to 24 hours.

Transactions

Amazon DynamoDB transactions simplify the developer experience of making coordinated, all-or-nothing changes to multiple items both within and across tables. *Transactions* provide atomicity, consistency, isolation, and durability (ACID) enabling you to maintain data correctness in your applications more easily.

You can use the DynamoDB transactional read and write APIs to manage complex business workflows that require adding, updating, or deleting multiple items as a single, all-or-nothing operation. With the transaction write API, you can group multiple `Put`, `Update`, `Delete`, and `ConditionCheck` actions. You can then submit the actions as a single `TransactWriteItems` operation that either succeeds or fails as a unit.

Related resource:

- DynamoDB Transactions: How it works - Explains how to group actions together and submit as all-or-nothing TransactWriteItems or TransactGetItems operations

PartiQL Query Access

Amazon DynamoDB supports PartiQL, a SQL-compatible query language, to select, insert, update, and delete data in Amazon DynamoDB. PartiQL can also be used to perform transactional operations.

You can run ad hoc PartiQL queries against tables. PartiQL operations provide the same availability, latency, and performance as the other DynamoDB data plane operations.

Related resources:

- PartiQL - a SQL-compatible query language for Amazon DynamoDB
- PartiQL Tutorial - learn how to write queries using the interactive shell, or REPL.

DynamoDB Accelerator (DAX) In-memory acceleration

In most cases, the DynamoDB response times can be measured in single-digit milliseconds. If your use case requires a response in microseconds, is read-heavy, or has bursty workloads, DAX provides fast response times for accessing eventually consistent data, increased throughput, and potential operational cost savings.

Related resource:

- In-memory acceleration with DynamoDB Accelerator (DAX) - for response times in microseconds, increased throughput, and potential operational cost savings for large read-heavy or bursty workloads.

Additional resources

Official AWS documentation:

- Amazon DynamoDB Developer Guide - extensive and complete documentation for Amazon DynamoDB
- Getting started with DynamoDB - tutorial to create a table, write/read/update and query data. You will use the AWS CLI, with the option to run PartiQL DB queries.

- Getting Started Resource Center - Choosing an AWS database service - Choosing the right database requires you to make a series of decisions based on your organizational needs. This decision guide will help you ask the right questions, provide a clear path for implementation, and help you migrate from your existing database.

Resources from the serverless community:

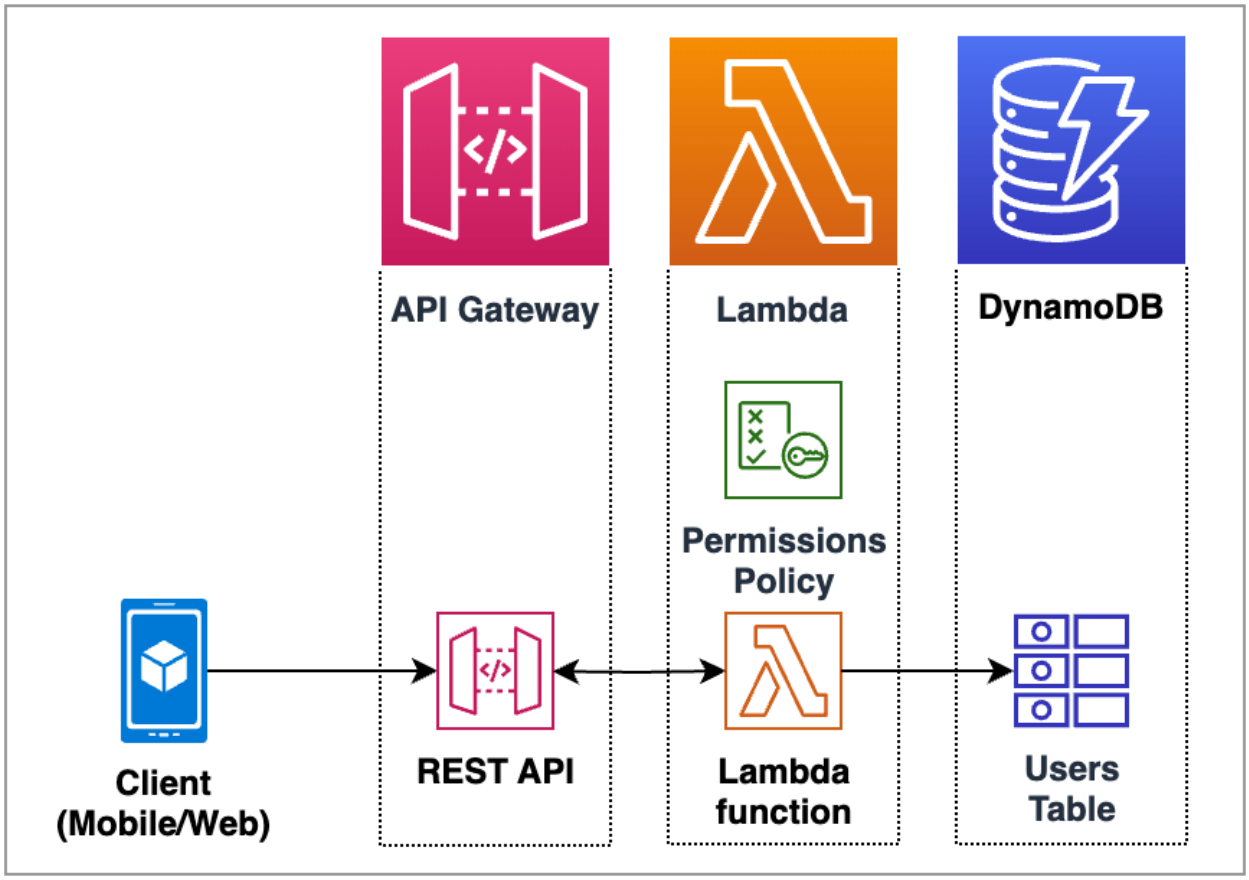
- Creating a single-table design with Amazon DynamoDB - blog article by James Beswick (26 JUL 2021) showing how to model many to one and many to many relationships with indexes in DynamoDB.
- Additional Amazon DynamoDB Resources - more links to blog posts, guides, presentations, training, and tools

Next steps

In parallel to this guide, a group of Amazon engineers are building a series of workshops based on architectural and design patterns that customers commonly use in real-world solutions. You get hands-on experience with infrastructure and code that you could actually deploy as part of a production solution.

Learn serverless techniques in an online workshop

Learn by doing in the **Serverless Patterns Workshop**. The first module introduces a serverless microservice to retrieve data from DynamoDB with Lambda and API Gateway. Additional modules provide practical examples using infrastructure as code to deploy resources, test, and build with common architectural patterns used in serverless solutions.

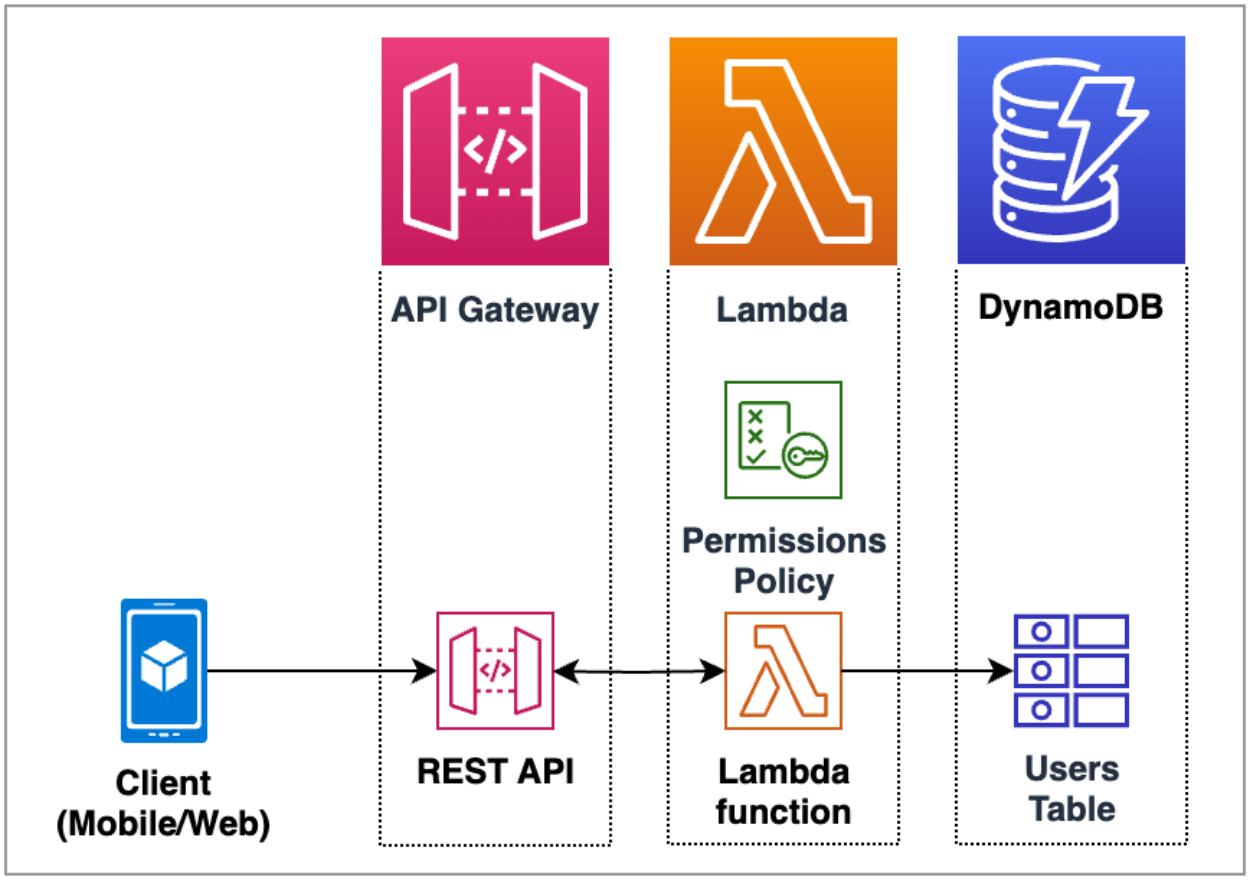


Learn using a workshop

In parallel to this guide, a group of Amazon engineers are building a series of workshops based on architectural and design patterns that customers commonly use in real-world solutions. You get hands-on experience with infrastructure and code that you could actually deploy as part of a production solution.

Learn serverless techniques in an online workshop

Learn by doing in the Serverless Patterns Workshop. The first module introduces a serverless microservice to retrieve data from DynamoDB with Lambda and API Gateway. Additional modules provide practical examples using infrastructure as code to deploy resources, test, and build with common architectural patterns used in serverless solutions.


API Gateway
Lambda
DynamoDB
Permissions Policy
Client (Mobile/Web)
REST API
Lambda function
Users Table

Document history for the Serverless Developer Guide

The following table describes notable releases to the Serverless Developer Guide.

Change	Description	Date
Minor revisions	Updated links to Serverless Patterns workshop (now with idempotence!). Fixed various links to additional resources.	August 28, 2023
Workshop connections	Added links to the related serverless workshop for hands-on experience.	April 12, 2023
Initial release	Initial release of the Serverles s Developer Guide!	February 19, 2023