AWS Serverless Multi-Tier Architectures

With Amazon API Gateway and AWS Lambda

September 2019

Notices

Contents

Abstract

This whitepaper illustrates how innovations from Amazon Web Services (AWS) can be used to change the way you design multi-tier architectures and implement popular patterns such as microservices, mobile back ends, and single page applications. Architects and developers can leverage Amazon API Gateway, AWS Lambda, and other services to reduce the development and operations cycles required to create and manage multi-tiered applications.

Introduction

The multi-tier application (three-tier, *n*-tier, and so on.) has been a cornerstone architecture pattern for decades, and remains a popular pattern for user-facing applications. Although the language used to describe a multi-tier architecture varies, a multi-tier application generally consists of the following components:

- **Presentation tier**: component that user directly interacts with (web page, mobile app UI, etc.)
- **Logic tier**: code required to translate user actions to application functionality (CRUD database operations, data processing, etc.)
- **Data tier**: storage media (databases, object stores, caches, file systems, etc.) that hold the data relevant to the application

The multi-tier architecture pattern provides a general framework to ensure decoupled and independently scalable application components that can be separately developed, managed, and maintained (often by distinct teams).

As a consequence of this pattern in which the network (a tier must make a network call to interact with another tier) acts as the boundary between tiers, developing a multi-tier application often requires creating many undifferentiated application components. Some of these components include:

- code that defines a message queue for communication between tiers,
- code that defines an API and a data model,
- and security-related code that ensures appropriate access to the application.

All of these examples can be considered “boilerplate” components that, while necessary in multi-tier applications, do not vary greatly in their implementation from one application to the next.

AWS offers a number of services that enable the creation of serverless multi-tier applications – greatly simplifying the process of deploying such applications to production and removing the overhead associated with traditional server management. Amazon API Gateway, a service for creating and managing APIs, and AWS Lambda, a service for running arbitrary code functions, can be used together to simplify the creation of robust multi-tier applications.

Amazon API Gateway's integration with AWS Lambda enables user-defined code functions to be triggered directly via HTTPS requests. Regardless of the request volume, both API Gateway and Lambda scale automatically to support exactly the needs of your application (see API Gateway Service Limits for scalability information). By combining these two services, you can create a tier that allows you to write only the code that matters to your application and not focus on various other undifferentiating aspects of implementing a multi-tiered architecture—like architecting for high availability, writing client SDKs, server/operating system (OS) management, scaling, and implementing a client authorization mechanism.

API Gateway and Lambda enable the creation of a serverless logic tier. Depending on your application requirements, AWS also provides options to create a serverless presentation tier (e.g., with Amazon CloudFront and Amazon S3) and data tier (e.g., Amazon Aurora, Amazon DynamoDB).

This whitepaper focuses on the most popular example of a multi-tiered architecture, the **three-tier** web application. However, you can apply this multi-tier pattern well beyond a typical three-tier web application.

Three-Tier Architecture Overview

The three-tier architecture is the most popular implementation of a multi-tier architecture and consists of a single presentation tier, logic tier, and data tier. *Figure 1* shows an example of a simple, generic three-tier application.

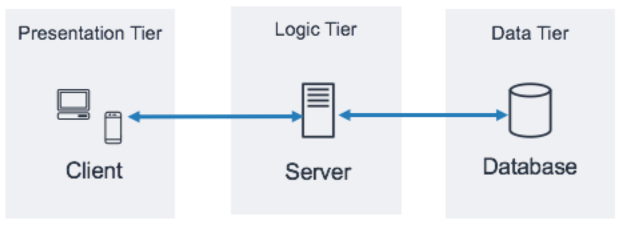

Figure 1: Architectural pattern for a simple three-tier application

There are many great resources online where you can learn more about the *general* three-tier architecture pattern. This whitepaper focuses on a specific implementation pattern for this architecture using Amazon API Gateway and AWS Lambda.

Serverless Logic Tier

The logic tier of the three-tier architecture represents the brains of the application. This is where using Amazon API Gateway and AWS Lambda can have the most impact compared to a traditional, server-based implementation. The features of these two services allow you to build a serverless application that is highly available, scalable, and secure. In a traditional model, your application could require thousands of servers; however, by leveraging Amazon API Gateway and AWS Lambda you are not responsible for server management in any capacity. In addition, by using these managed services together, you gain the following benefits:

- AWS Lambda:
  - No operating systems to choose, secure, patch, or manage.
  - No servers to right size, monitor, or scale.
  - Reduced risk to your cost from over-provisioning
  - Reduced risk to your performance from under-provisioning.
- Amazon API Gateway:
  - Simplified mechanisms to deploy, monitor, and secure APIs.
  - Improved API performance via caching and content delivery

AWS Lambda

AWS Lambda is a compute service that allows you to run arbitrary code functions in any of the supported languages (Node.js, Python, Ruby, Java, Go, .NET, for more information, see Lambda FAQs) without provisioning, managing, or scaling servers. Lambda functions are executed in a managed, isolated container and are triggered in response to an event which can be one of several programmatic triggers that AWS makes available, called an **event source** (see Lambda FAQs for all event sources).

Many popular use cases for AWS Lambda revolve around event-driven data processing workflows, such as processing files stored in Amazon Simple Storage Service (Amazon S3) or streaming data records from Amazon Kinesis. When used in conjunction with Amazon API Gateway, an AWS Lambda function performs the functionality of a typical web service: it executes code in response to a client HTTPS request. Amazon API Gateway acts as the front door for your logic tier, and AWS Lambda executes the application code.

Your Business Logic Goes Here, No Servers Necessary

AWS Lambda requires you to write code functions, called handlers, which will execute when triggered by an event. To use AWS Lambda with Amazon API Gateway, you can configure API Gateway to trigger handler functions when an HTTPS request to your API occurs. In a serverless multi-tier architecture, each of the APIs you create in Amazon API Gateway will integrate with a Lambda function (and the handler within) that executes the business logic required.

Using AWS Lambda functions to compose the logic tier allows you to define a desired level of granularity for exposing the application functionality (one Lambda function per API or one Lambda function per API method). Inside the Lambda function, the handler is free to reach out to any other dependencies (for example, other methods you've uploaded with your code, libraries, native binaries, and external web services), or even other Lambda functions.

Creating or updating a Lambda function requires creating a *Lambda deployment package*, which involves bundling all required dependencies into a static archive (that is, .zip) during creation. When you create your function in the console or via the AWS Command Line Interface (AWS CLI), you specify which method inside your deployment package will act as the request handler. You are free to reuse the same deployment package for multiple Lambda function definitions, where each Lambda function may have a unique handler within the same deployment package.

Lambda Security

To execute a Lambda function, it must be triggered by an event or service that is permitted to do so via an AWS Identity and Access Management (IAM) policy. Using IAM policies, you can create a Lambda function that cannot be executed *at all* unless it is invoked by an API Gateway resource that you define.

Each Lambda function itself assumes an IAM role that is assigned when the Lambda function is deployed. This IAM role defines the other AWS services and resources your Lambda function can interact with (for example, Amazon DynamoDB table, Amazon S3).

You should not store sensitive information inside a Lambda function. AWS IAM handles access to AWS services via Lambda execution roles; if you need to access other credentials (for example, database credentials, API Keys) from inside your Lambda function, you can use AWS Key Management Service (AWS KMS) with environment variables, or use a service like AWS Secrets Manager to keep this information safe when not in use.

Performance at Scale

Code uploaded to AWS Lambda is stored in Amazon S3 and runs in an isolated environment managed by AWS. You do not have to scale your Lambda functions – each time an event notification is received by your function, AWS Lambda locates free capacity within its compute fleet and runs your code with runtime, memory, disk, and timeout configurations that you define. With this pattern, AWS can start as many copies of your function as needed.

A Lambda-based logic tier is always right sized for your customer needs. The ability to quickly absorb surges in traffic through managed scaling and concurrent executions, combined with Lambda pay-per-use pricing, enables you to always meet customer requests while simultaneously not paying for idle compute capacity.

Serverless Deployment and Management

To help you deploy and manage your Lambda functions, use the AWS Serverless Application Model (AWS SAM), an open-source framework that includes:

- **SAM Template Specification:** Syntax used to define your functions and describe their environments, permissions, configurations, and events for simplified upload and deployment
- **SAM Command Line Interface (CLI):** Commands that enable you to verify SAM template syntax, invoke functions locally, debug Lambda functions, and package/deploy functions

Normally, when you deploy a Lambda function, it is executed with permissions defined by its assigned IAM role, and is able to reach internet-facing endpoints. As the core of your logic tier, AWS Lambda is the component directly integrating with the data tier. If your data tier contains sensitive business or user information, it is important to ensure that this data tier is appropriately isolated (in a private subnet).

For your Lambda function to access resources that you cannot expose publicly, like a private database instance, you can place your AWS Lambda function inside an Amazon Virtual Private Cloud (Amazon VPC) and configure an Elastic Network Interface (ENI) to access your internal resources.

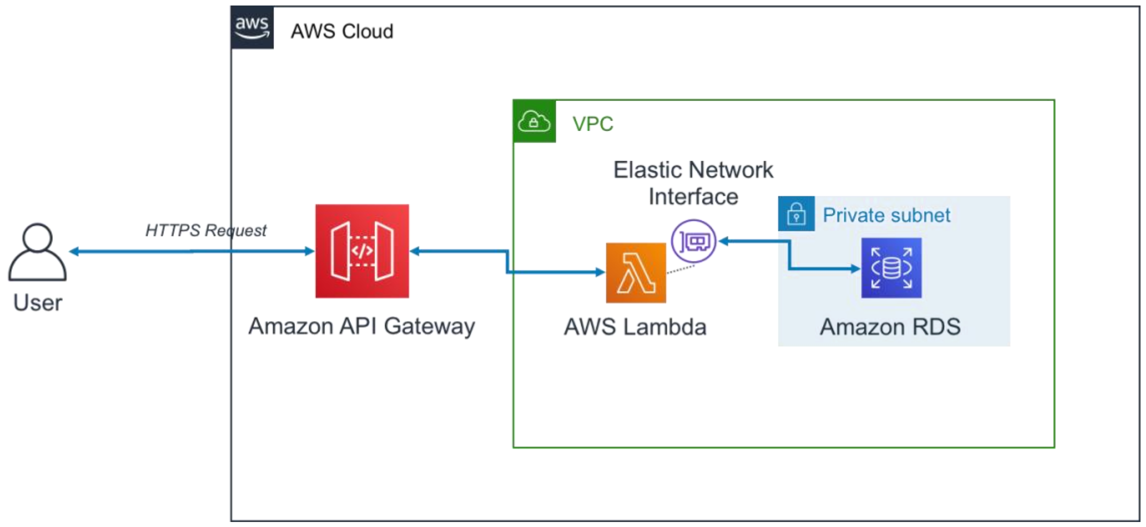

Figure 2: Lambda architecture pattern inside a VPC

The use of Lambda with VPC means that databases and other storage media that your business logic depends on can be made inaccessible over the internet. The VPC also ensures that the only way to interact with your data from the internet is through the APIs that you've defined and the Lambda code functions that you have written.

Amazon API Gateway

Amazon API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. Clients (that is, presentation tiers) integrate with the APIs exposed via API Gateway using standard HTTPS requests. The applicability of APIs exposed via API Gateway to a service-oriented multi-tier architecture is the ability to separate individual pieces of application functionality and expose this functionality via REST endpoints. Amazon API Gateway has specific features and qualities that can add powerful capabilities to your logic tier.

Integration with AWS Lambda

An Amazon API Gateway REST API is made up of resources and methods. A resource is a logical entity that an app can access through a resource path (e.g., `/tickets`). A method corresponds to a REST API request that is submitted to an API resource (e.g., `GET /tickets`). API Gateway allows you to back each method with a Lambda function, that is, when you call the API through the HTTPS endpoint exposed in API Gateway, API Gateway invokes the Lambda function.

You can connect API Gateway and Lambda functions using Proxy Integrations and Non-Proxy Integrations.

Proxy Integrations

In a Proxy Integration, the entire client HTTPS request is sent to the Lambda function "as-is." API Gateway passes the entire client request as the `event` parameter of the Lambda handler function, and the output of the Lambda function is returned directly to the client (including status code, headers, and so on.).

Non-Proxy Integrations

In a Non-Proxy Integration, you configure how the parameters, headers, and body of the client request are passed to the `event` parameter of the Lambda handler function. Additionally, you configure how the Lambda output is translated back to the user.

Note that API Gateway can also proxy to additional serverless resources outside AWS Lambda, such as mock integrations (useful for initial application development), and direct proxy to S3 objects.

Stable API Performance Across Regions

Each deployment of Amazon API Gateway includes an Amazon CloudFront distribution under the hood. Amazon CloudFront is a content delivery service that uses Amazon's global network of edge locations as connection points for clients using your API. This helps decrease the response latency of your API. By using multiple edge locations across the world, Amazon CloudFront also provides capabilities to combat distributed denial of service (DDoS) attack scenarios. For more information– read the AWS Best Practices for DDoS Resiliency whitepaper.

You can improve the performance of specific API requests by using Amazon API Gateway to store responses in an optional in-memory cache. This approach not only provides performance benefits for repeated API requests, but it also reduces the number of times your Lambda functions are executed, which can reduce your overall cost.

Encourage Innovation and Reduce Overhead with Built-In Features

The development cost to build any new application is an investment. Using Amazon API Gateway can reduce the amount of time required for certain development tasks and lower the total development cost, allowing organizations to more freely experiment and innovate.

During initial application development phases, implementation of logging and metrics gathering are often neglected to deliver a new application more quickly. This can lead to technical debt and operational risk when deploying these features to an application running in production. Amazon API Gateway integrates seamlessly with Amazon CloudWatch, which collects and processes raw data from API Gateway into readable, near real-time metrics for monitoring API execution. API Gateway also supports access logging with configurable reports, and AWS X-Ray tracing for debugging. Each of these features requires no code to be written, and can be adjusted in applications running in production without risk to the core business logic.

The overall lifetime of an application may be unknown, or it may be known to be short-lived. Creating a business case for building such applications can be made easier if your starting point already includes the managed features that Amazon API Gateway provides, and if you only incur infrastructure costs after your APIs begin receiving requests. For more information, see Amazon API Gateway pricing.

Iterate Rapidly, Stay Agile

Using Amazon API Gateway and AWS Lambda to build the logic tier of your API enables you to quickly adapt to the changing demands of your user base by simplifying API deployment and version management.

When you deploy an API in API Gateway, you must associate the deployment with an API Gateway stage – each stage is a snapshot of the API and is made available for client apps to call. Using this convention, you can easily deploy apps to *dev*, *test*, *stage*, or *prod* stages, and move deployments between stages. Each time you deploy your API to a stage, you create a different version of the API which can be rolled back if necessary. These features enable existing functionality and client dependencies to continue undisturbed while new functionality is released as a separate API/function version.

API Gateway offers additional deployment features that can be used to simplify publication of APIs such as canary release deployments and custom domain names.

Prioritize API Security

All applications must ensure that only authorized clients have access to their API resources. When designing a multi-tier application, you can take advantage of several different ways in which Amazon API Gateway contributes to securing your logic tier:

Transit Security

All requests to your APIs can be made via HTTPS to enable encryption in transit.

API Gateway provides built-in SSL/TLS Certificates – if using the custom domain name option for public-facing APIs, you can provide your own SSL/TLS certificate using AWS Certificate Manager.

API Authorization

Each resource/method combination that you create as part of your API is granted its own specific Amazon Resource Name (ARN) that can be referenced in AWS IAM policies.

There are three general methods to add Authorization to an API in Amazon API Gateway:

- **IAM Roles and Policies:** Clients use AWS Signature Version 4 (SigV4) authorization and IAM policies for API access. The same credentials can restrict or permit access to other AWS services and resources as needed (for example, Amazon S3 buckets or Amazon DynamoDB tables).
- **Amazon Cognito User Pools:** Clients sign in via an Amazon Cognito user pool and obtain tokens, which are included in the Authorization header of a request.
- **Custom Authorizer:** Define a Lambda function that implements a custom authorization scheme that uses a bearer token strategy (for example, OAuth, SAML) or uses request parameters to identify users.

Access Restrictions

Amazon API Gateway supports generation of API Keys and association of these keys with a configurable usage plan. You can monitor API Key usage with Amazon CloudWatch.

API Gateway supports throttling, rate limits, and burst rate limits for each method in your API.

You can also use AWS WAF (Web Application Firewall) with Amazon API Gateway to help protect APIs from attack.

Data Tier

Using AWS Lambda as your logic tier does not limit the data storage options available in your data tier. Lambda functions connect to any data storage option by including the appropriate database driver in the Lambda deployment package, and use IAM role-based access or encrypted credentials (via AWS KMS or AWS Secrets Manager).

Choosing a data store for your application is highly dependent on your application requirements. AWS offers a number of serverless and non-serverless data stores that you can use to compose the data tier of your application.

Serverless Data Storage Options

Amazon S3 an object storage service that offers industry-leading scalability, data availability, security, and performance.

Amazon Aurora is a MySQL and PostgreSQL-compatible relational database built for the cloud, that combines the performance and availability of traditional enterprise databases with the simplicity and cost-effectiveness of open source databases. Amazon Aurora offers both serverless and traditional usage models.

Amazon DynamoDB is a key-value and document database that delivers single-digit millisecond performance at any scale. It is a fully managed, serverless, multi-region, multi-master, durable database with built-in security, backup and restore, and in-memory caching for internet-scale applications.

Amazon Timestream is a fast, scalable, fully managed time series database service for IoT and operational applications that makes it easy to store and analyze trillions of events per day at 1/10th the cost of relational databases. Driven by the rise of IoT devices, IT systems, and smart industrial machines, time-series data—data that measures how things change over time—is one of the fastest growing data types.

Amazon Quantum Ledger Database (Amazon QLDB) is a fully managed ledger database that provides a transparent, immutable, and cryptographically verifiable transaction log owned by a central trusted authority. Amazon QLDB tracks each and every application data change and maintains a complete and verifiable history of changes over time.

Non-Serverless Data Storage Options

Amazon Relational Database Service (Amazon RDS) is a managed web service that makes it easier to set up, operate, and scale a relational database using any of the available engines (Amazon Aurora, PostgreSQL, MySQL, MariaDB, Oracle, and SQL Server) and running on several different database instance types that are optimized for memory, performance, or I/O.

Amazon Redshift is a fully managed, petabyte-scale data warehouse service in the cloud.

Amazon ElastiCache is a fully managed deployment of Redis or Memcached. Seamlessly deploy, run, and scale popular open source compatible in-memory data stores.

Amazon Neptune is a fast, reliable, fully managed graph database service that makes it easy to build and run applications that work with highly connected datasets. Amazon Neptune supports popular graph models Property Graph and W3C's RDF, and their respective query languages allowing you to easily build queries that efficiently navigate highly connected datasets.

Amazon DocumentDB (with MongoDB compatibility) is a fast, scalable, highly available, and fully managed document database service that supports MongoDB workloads.

Finally, you can also use data stores running independently on Amazon EC2 as the data tier of a multi-tier application

Presentation Tier

The presentation tier is responsible for interacting with the logic tier via the API Gateway REST endpoints exposed over the internet. Any HTTPS capable client or device can communicate with these endpoints, giving your presentation tier the flexibility to take many forms (desktop applications, mobile apps, web pages, IoT devices, and so on). Depending on your requirements, your presentation tier can use the following AWS serverless offerings:

- Amazon Cognito is a serverless user identity and data synchronization service that allows you to add user sign-up, sign-in, and access control to your web and mobile apps quickly and easily. Amazon Cognito scales to millions of users and supports sign-in with social identity providers, such as Facebook, Google, and Amazon, and enterprise identity providers via SAML 2.0.
- Amazon S3 with Amazon CloudFront: Amazon S3 allows you to serve static websites, such as single page applications (SPAs), directly from an S3 bucket without requiring provision of a web server. You can use Amazon CloudFront as a managed content delivery network (CDN) to improve performance and enable SSL/TL using managed or custom certificates.

Depending on your networking configurations and application requirements, you may need to enable your Amazon API Gateway APIs to be cross-origin resource sharing (CORS)-compliant. CORS compliance allows web browsers to directly invoke your APIs from within the static web pages.

When you deploy a website with Amazon CloudFront, you are provided a CloudFront domain name to reach your application (for example, d2d47p2vcczkh2.cloudfront.net). You can use Amazon Route 53 to register domain names and direct them to your CloudFront distribution, or direct already-owned domain names to your CloudFront distribution. This allows users to access your site using a familiar domain name. Note that you can also assign a custom domain name using Route 53 to your API Gateway distribution, which allows users to invoke APIs using familiar domain names.

Sample Architecture Patterns

You can implement popular architecture patterns using Amazon API Gateway and AWS Lambda as your logic tier. This whitepaper includes the most popular architecture patterns that leverage AWS Lambda based logic tiers:

- **Mobile Backend:** A mobile application communicates with API Gateway and Lambda to access application data.

  This pattern can be extended to generic HTTPS clients that are not using serverless AWS resources to host presentation tier resources (for example, desktop clients, web server running on EC2, and so on).

- **Single Page Application:** A single page application hosted in Amazon S3 and Amazon CloudFront communicates with API Gateway and Lambda to access application data.

In addition to these two patterns, we discuss the applicability of Lambda and API Gateway to a general microservice architecture. A microservice architecture is a popular pattern that, although is not a standard three-tier architecture, involves decoupling application components and deploying them as stateless, individual units of functionality that communicate with each other.

Mobile Backend

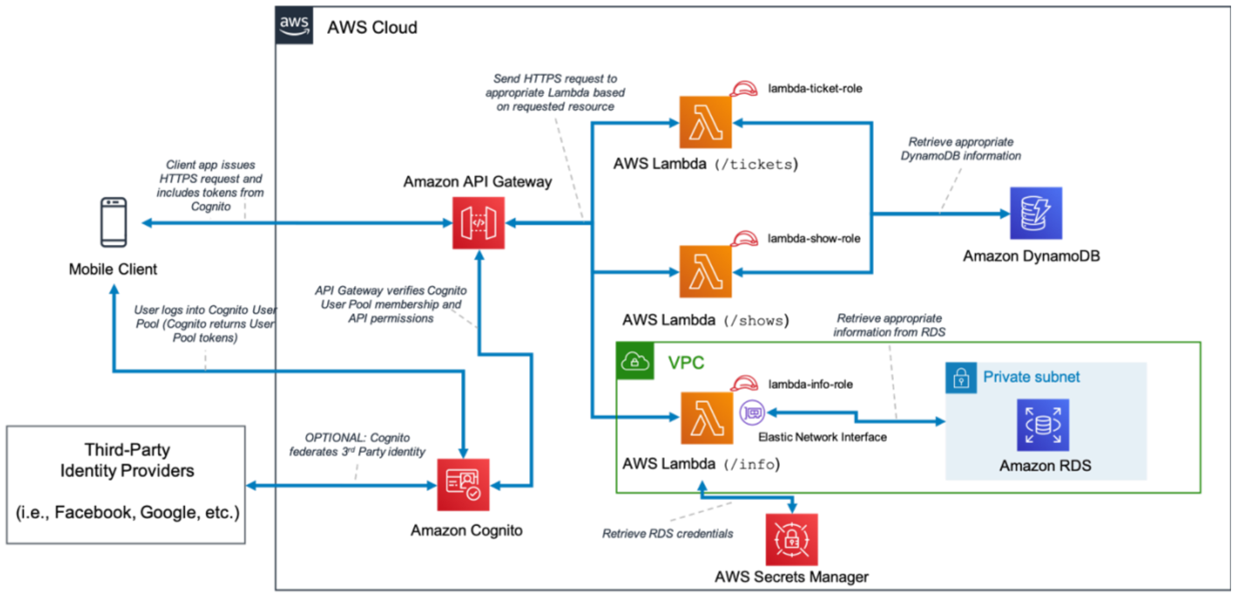
AWS Cloud
Send HTTPS request to appropriate Lambda based on requested resource
lambda-ticket-role
AWS Lambda (/tickets)
Retrieve appropriate DynamoDB information
Client app issues HTTPS request and includes tokens from Cognito
Amazon API Gateway
Mobile Client
lambda-show-role
Amazon DynamoDB
API Gateway verifies Cognito User Pool membership and API permissions
User logs into Cognito User Pool (Cognito returns User Pool tokens)
AWS Lambda (/shows)
Retrieve appropriate information from RDS
VPC
Private subnet
lambda-info-role
Elastic Network Interface
Amazon RDS
Third-Party Identity Providers
(i.e., Facebook, Google, etc.)
OPTIONAL: Cognito federates 3rd Party identity
AWS Lambda (/info)
Amazon Cognito
Retrieve RDS credentials
AWS Secrets Manager

Figure 3: Architectural pattern for serverless mobile backend

Table 1: Mobile backend tier components

Tier	Components
Presentation	Mobile application running on a user device.
Logic	Amazon API Gateway with AWS Lambda. This architecture shows three exposed services (`/tickets`, `/shows`, and `/info`). API Gateway endpoints are secured via Amazon Cognito user pools. In this method, users sign in to Amazon Cognito user pools (using a federated third-party if necessary), and receive access and id tokens that are used to authorize Amazon API Gateway calls. Each Lambda function is assigned its own IAM role to provide access to the appropriate data source.
Data	Amazon DynamoDB is used for the `/tickets` and `/shows` services Amazon RDS is used for the `/info` service. This Lambda function retrieves Amazon RDS credentials from AWS Secrets Manager and uses an Elastic Network Interface to access the private subnet.

Single Page Application

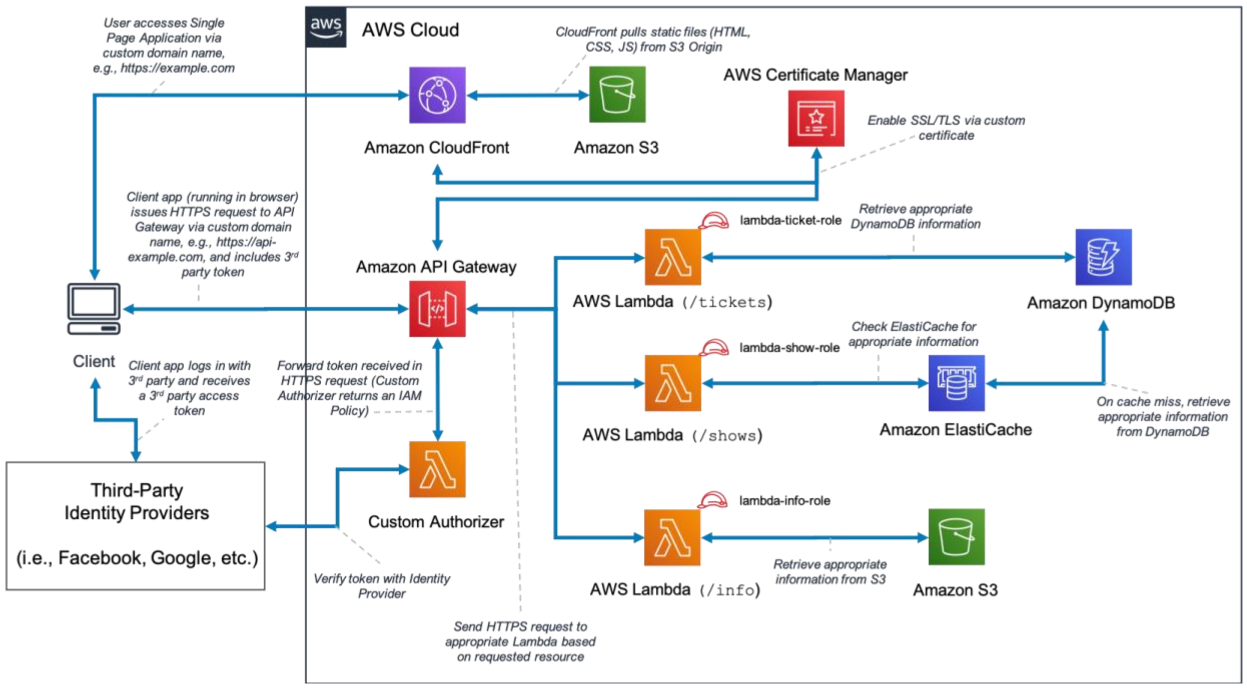
User accesses Single Page Application via custom domain name, e.g., https://example.com
AWS Cloud
CloudFront pulls static files (HTML, CSS, JS) from S3 Origin
Amazon CloudFront
Amazon S3
AWS Certificate Manager
Enable SSL/TLS via custom certificate
Client app (running in browser) issues HTTPS request to API Gateway via custom domain name, e.g., https://api-example.com, and includes 3rd party token
Amazon API Gateway
lambda-ticket-role
Retrieve appropriate DynamoDB information
AWS Lambda (/tickets)
Amazon DynamoDB
Client
Client app logs in with 3rd party and receives a 3rd party access token
Forward token received in HTTPS request (Custom Authorizer returns an IAM Policy)
lambda-show-role
Check ElastiCache for appropriate information
AWS Lambda (/shows)
Amazon ElastiCache
On cache miss, retrieve appropriate information from DynamoDB
Third-Party Identity Providers
(i.e., Facebook, Google, etc.)
Custom Authorizer
lambda-info-role
AWS Lambda (/info)
Retrieve appropriate information from S3
Amazon S3
Verify token with Identity Provider
Send HTTPS request to appropriate Lambda based on requested resource

Figure 4: Architectural pattern for serverless single page application

Table 2: Single page application tier components

Tier	Components
Presentation	Static website content hosted in Amazon S3, distributed by Amazon CloudFront. AWS Certificate Manager allows a custom SSL/TLS certificate to be used.
Logic	Amazon API Gateway with AWS Lambda. This architecture shows three exposed services (`/tickets`, `/shows`, and `/info`). API Gateway endpoints are secured via a Custom Authorizer. In this method, users sign in via a third-party Identity Provider and obtain access and id tokens. These tokens are included in API Gateway calls, and the Custom Authorizer validates these tokens and generates an IAM policy containing API execution permissions. Each Lambda function is assigned its own IAM role to provide access to the appropriate data source.

Tier	Components
Data	Amazon DynamoDB is used for the `/tickets` and `/shows` services. Amazon ElastiCache is used by the `/shows` service to improve database performance. Cache misses are sent to DynamoDB. Amazon S3 is used to host static content used by the `/info` service.

Microservices with Lambda

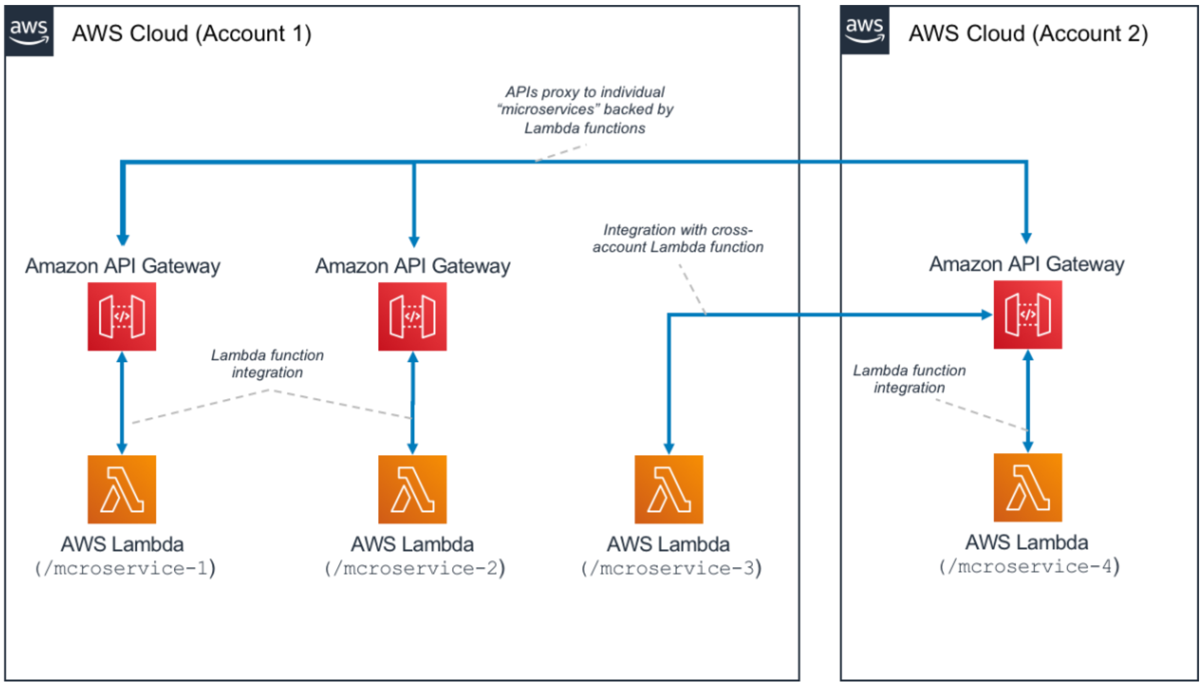
AWS Cloud (Account 1)
AWS Cloud (Account 2)
APIs proxy to individual "microservices" backed by Lambda functions
Integration with cross-account Lambda function
Amazon API Gateway
Amazon API Gateway
Amazon API Gateway
Lambda function integration
Lambda function integration
AWS Lambda (/mcroservice-1)
AWS Lambda (/mcroservice-2)
AWS Lambda (/mcroservice-3)
AWS Lambda (/mcroservice-4)

Figure 5: Architectural pattern for microservices with Lambda

The microservice architecture pattern is not bound to the typical three-tier architecture; however, this popular pattern can realize significant benefits from the use of serverless resources.

In this architecture, each of the application components are decoupled and independently deployed and operated. An API created with Amazon API Gateway, and functions subsequently executed by AWS Lambda, is all that you need to build a microservice. Your team is free to use these services to decouple and fragment your environment to the level of granularity desired.

In general, a microservices environment can introduce the following difficulties: repeated overhead for creating each new microservice, issues with optimizing server density/utilization, complexity of running multiple versions of multiple microservices simultaneously, and proliferation of client-side code requirements to integrate with many separate services.

When you create microservices using serverless resources, these problems become simpler to solve and, in some cases, simply disappear. The serverless microservices pattern lowers the barrier for the creation of each subsequent microservice (Amazon API Gateway even allows for the cloning of existing APIs, and usage of Lambda functions in other accounts). Optimizing server utilization is no longer relevant with this pattern. Finally, Amazon API Gateway provides programmatically generated client SDKs in a number of popular languages to reduce integration overhead.

Conclusion

The multi-tier architecture pattern encourages the best practice of creating application components that are easy to maintain, decouple, and scale. When you create a logic tier where integration occurs via Amazon API Gateway and computation occurs within AWS Lambda, you realize these goals while reducing the amount of effort to achieve them. Together, these services provide a HTTPS API front end for your clients and a secure environment to execute your business logic all while removing the overhead involved with managing typical server-based infrastructure.

Contributors

Contributors to this document include:

- Andrew Baird, AWS Solutions Architect
- Bryant Bost, AWS ProServe Consultant
- Stefano Buliani, Senior Product Manager, Tech, AWS Mobile
- Vyom Nagrani, Senior Product Manager, AWS Mobile
- Ajay Nair, Senior Product Manager, AWS Mobile

Further Reading

For additional information, see:

- AWS Whitepapers page

Document Revisions

Date	Description
September 2019	Updated for new service features.
November 2015	First publication.