Serverless Architectures

- Serverless architectures allow developers to build and run applications without managing servers.
- The cloud provider automatically handles:
    - Server provisioning
    - Scaling
    - High availability
    - Maintenance
- Applications are composed of managed services that integrate with each other, such as:
    - AWS Lambda
    - API Gateway
    - DynamoDB
    - S3
    - Step Functions
- Developers focus purely on writing code and configuring service interactions.

Mobile Application: MyTodoList

We want to build a mobile application with the following requirements:

- The backend must be exposed as a **REST API over HTTPS**
- Architecture must be **serverless**
- Users should be able to **directly interact with their own folder in S3**
- **User authentication** must be handled through a **managed serverless service**
- Users can **write and read to-dos**, but the application is **read-heavy**
- The **database** must be **scalable** and support **high read throughput**

Mobile App: REST API Layer

Architecture Overview

1. **Mobile client** sends REST HTTPS requests to **Amazon API Gateway**.
2. **Amazon Cognito** handles user **authentication**.
3. API Gateway **verifies the authentication** using Cognito.
4. Once authenticated, the request is forwarded to **AWS Lambda**.
5. Lambda invokes **Amazon DynamoDB** to perform read/write operations (e.g., querying to-dos).

This setup ensures secure, serverless, and scalable interactions between the mobile app and AWS services.

Mobile App: Giving Users Access to S3

Architecture Overview

1. **Mobile client** authenticates through **Amazon Cognito**.
2. Upon authentication, **temporary AWS credentials** are granted via Cognito Identity Pools.
3. These credentials allow the client to **directly store and retrieve files** in **Amazon S3**, based on user-specific permissions.
4. **Amazon API Gateway** and **AWS Lambda** are still used for other backend logic (e.g., managing metadata in **Amazon DynamoDB**).

This setup enables secure, user-specific access to S3 while maintaining a fully serverless architecture.

Mobile App: High Read Throughput, Static Data

Architecture Overview

1. **Mobile client** authenticates via **Amazon Cognito**.
2. Once authenticated, the client sends REST HTTPS requests through **Amazon API Gateway**.
3. API Gateway verifies the authentication and invokes **AWS Lambda**.
4. Lambda queries **DynamoDB**, which is backed by **DAX** (DynamoDB Accelerator) to improve **read throughput**.
5. For static or less frequently changing data, **Amazon S3** is used to store and serve files.
6. The client can directly **store/retrieve files** from S3 using the permissions granted after Cognito authentication.

This setup is optimized for applications with **read-heavy workloads** and **static content delivery**.

Mobile App: Caching at the API Gateway

Architecture Overview

1. **Mobile client** authenticates via **Amazon Cognito**.
2. Requests are sent via **REST HTTPS** to **Amazon API Gateway**.
3. **API Gateway** verifies the authentication and can cache **responses** to reduce backend load.
4. Cached responses reduce invocations of **AWS Lambda** and queries to **DynamoDB**.
5. For dynamic queries, **DynamoDB** can still be queried directly with acceleration provided by **DAX**.
6. **Amazon S3** is used for storing and retrieving static files directly from the mobile client.

Benefit

- Caching at API Gateway level improves performance and reduces cost for **read-heavy workloads**.

In This Lecture

Topics Covered

- Building a **serverless REST API** using:
    - HTTPS
    - Amazon API Gateway
    - AWS Lambda
    - Amazon DynamoDB
- Using **Amazon Cognito** to generate **temporary AWS credentials**:
    - Allows users to directly access **S3 buckets** with **restricted IAM policies**
    - This pattern can be extended to other services like **DynamoDB**, **Lambda**, etc.
- **Read caching** on DynamoDB using **DAX (DynamoDB Accelerator)**
- **REST request caching** at the **API Gateway** level
- Implementing **security**:
    - **Authentication and authorization** using Cognito

Serverless Hosted Website: MyBlog.com

Requirements

- The website must **scale globally**
- Blog posts are **rarely written** but **frequently read**
- The site consists of:
  - **Static files** (e.g., HTML, CSS, JS)
  - A **dynamic REST API** for interactive features
- **Caching** should be implemented wherever possible to improve performance
- When a **new user subscribes**, they should receive a **welcome email**
- When a **photo is uploaded**, a **thumbnail** should be automatically generated

Serving Static Content, Globally

Architecture Overview

- **Amazon S3** hosts the static website content (HTML, CSS, JS, images).
- **Amazon CloudFront** is used as a **Content Delivery Network (CDN)** to distribute the content globally.
- Clients interact with the nearest **CloudFront edge location**, which:
  - Reduces latency
  - Improves load times
  - Lowers origin load on S3

This setup ensures fast and scalable delivery of static assets across the world.

Serving Static Content, Globally, Securely

Secure Global Distribution with CloudFront

- **Amazon S3** hosts the static content.
- **Amazon CloudFront** handles global **content distribution** through edge locations for low-latency access.
- **Client** interacts only with **CloudFront edge locations**.

Security with Origin Access Control (OAC)

- **OAC** restricts access to the S3 bucket:
  - S3 bucket is **not publicly accessible**
  - Only the **CloudFront Distribution** is authorized to access the bucket
- Enforced via **S3 bucket policy**

This configuration ensures that static content is served securely and efficiently to users worldwide.

Adding a Public Serverless REST API

Static Content Delivery

- **Amazon S3** stores static assets (e.g., HTML, JS, CSS).
- **Amazon CloudFront** globally distributes static content via edge locations.
- Access to the S3 bucket is restricted using **Origin Access Control (OAC)**:
  - Only requests from **CloudFront** are authorized.

REST API Integration

- A **public REST API** is added using:
    - **Amazon API Gateway** to expose the API over HTTPS
    - **AWS Lambda** to handle backend logic
    - **Amazon DynamoDB** as the database
    - **DAX (DynamoDB Accelerator)** to cache and speed up read operations

Client Interaction

- The **client** interacts with:
    - **CloudFront** for static content
    - **API Gateway** for dynamic REST API requests

This architecture enables secure, scalable delivery of both static and dynamic content.

Leveraging DynamoDB Global Tables

Static Content Delivery

- **Amazon S3** stores the website's static content.
- **Amazon CloudFront** distributes content globally via edge locations.
- **Origin Access Control (OAC)** ensures only CloudFront can access the S3 bucket.

Dynamic REST API Layer

- **Amazon API Gateway** exposes REST endpoints.
- **AWS Lambda** handles request processing logic.
- **Amazon DynamoDB Global Tables** ensure:
    - **Multi-region** replication of data
    - **Low-latency** read/write access for global users

Caching

- **DAX (DynamoDB Accelerator)** provides high-performance read caching.

Client Interaction

- Clients interact with CloudFront for static content and with API Gateway for dynamic requests, benefiting from both performance and global data consistency.

This setup is ideal for globally distributed applications that require fast, localized access to both static and dynamic content.

User Welcome Email Flow

Static Content Delivery

- **Amazon S3** hosts static assets.
- **Amazon CloudFront** distributes them globally.
- **OAC (Origin Access Control)** restricts access to S3 only via CloudFront.

REST API Layer

- **Amazon API Gateway** handles HTTPS requests.
- **AWS Lambda** processes requests.
- **Amazon DynamoDB** stores user data.
- **DAX** is used to cache frequent reads.

Welcome Email Workflow

1. When a user signs up, a new record is added to **DynamoDB**.
2. **DynamoDB Streams** capture this data change event.
3. The event triggers an **AWS Lambda** function.
4. Lambda uses the **AWS SDK** to call **Amazon SES (Simple Email Service)**.
5. An **IAM Role** grants the Lambda permission to send the email.

This flow ensures automatic delivery of a welcome email upon user registration in a fully serverless manner.

Thumbnail Generation Flow

Static Content and API

- **Amazon S3** hosts static files.
- **Amazon CloudFront** serves static content globally.
- **OAC (Origin Access Control)** restricts direct access to S3, allowing only CloudFront.
- **Amazon API Gateway** exposes the REST API.
- **AWS Lambda** and **DynamoDB** handle backend logic.
- **DAX** provides high-speed caching for DynamoDB queries.

Thumbnail Generation Process

1. **Client** uploads a photo to **Amazon S3** (with optional **Transfer Acceleration** enabled).
2. The **upload** triggers an **event** in the S3 bucket.
3. The event invokes an **AWS Lambda** function.
4. The Lambda function:
   - Generates a **thumbnail**
   - Stores it back in **Amazon S3**
5. Optionally, the flow can include:
   - **SQS** to decouple processing steps
   - **SNS** to notify other systems or trigger further actions

This architecture supports automatic image processing in a serverless and scalable way.

AWS Hosted Website Summary

Key Components and Architecture

- **Static content** was distributed globally using:
  - **Amazon S3** as the storage layer
  - **Amazon CloudFront** for CDN distribution
- The **REST API** was implemented using a **serverless architecture**:
  - **No Cognito** was required since the API was public
- **Amazon DynamoDB Global Tables** were used for:
  - Multi-region data replication
  - Low-latency global data access
  - *(Note: Aurora Global Database could be an alternative)*
- **DynamoDB Streams** were enabled to trigger **AWS Lambda**

- The **Lambda function**:
  - Was assigned an **IAM role**
  - Used **Amazon SES (Simple Email Service)** to send welcome emails
- **Amazon S3** was configured to trigger:
  - **SQS** (Simple Queue Service)
  - **SNS** (Simple Notification Service)
  - **Lambda** functions

This summary reflects a fully serverless, globally distributed architecture with automated notifications and processing.

Microservices Architecture

Key Goals

- Transitioning to a **microservices-based architecture**

Characteristics

- Multiple services interact directly via **REST APIs**
- Each microservice can have a **different architecture**, tailored to its specific needs
- Architecture can vary in:
  - Technology stack
  - Deployment model
  - Storage choice
  - Scalability strategy

Objective

- Enable a **leaner development lifecycle**:
  - Teams can work independently on different services
  - Faster iterations and deployments
  - Improved modularity and maintainability

Microservices Environment

User Access and Routing

- **Users** access individual services via subdomains:
  - `service1.example.com`
  - `service2.example.com`
  - `service3.example.com`
- **Amazon Route 53** handles DNS queries for service discovery
- Communication occurs over **HTTPS**

Service Infrastructure Components

Each service can be backed by a unique stack depending on its needs:

- **Amazon API Gateway** for exposing REST APIs
- **AWS Lambda** for serverless compute
- **Amazon ElastiCache** for in-memory caching
- **Elastic Load Balancing** for traffic distribution

- **Amazon EC2** with **Auto Scaling** for traditional compute
- **Amazon RDS** for relational databases
- **Amazon ECS** for containerized workloads
- **Amazon DynamoDB** for NoSQL databases

This flexible environment supports a robust, scalable, and modular microservices architecture.

Discussions on Microservices

Design Flexibility

- Each microservice can be **designed independently**
- Support for both:
  - **Synchronous patterns**: API Gateway, Load Balancers
  - **Asynchronous patterns**: SQS, Kinesis, SNS, Lambda triggers (e.g., S3 events)

Challenges with Microservices

- Overhead in creating and managing each microservice
- Difficulty in optimizing **server density and utilization**
- Complexity from managing **multiple versions** of services simultaneously
- Increased **client-side integration** complexity due to many separate APIs

How Serverless Helps

- **API Gateway** and **Lambda**:
  - **Automatically scale**
  - **Cost-efficient** (pay-per-use model)
- Easier to **clone APIs** and reproduce environments
- **Client SDKs** can be generated automatically via **Swagger integration** with API Gateway

This makes serverless a compelling approach to reduce operational complexity in microservices architectures.

Software Updates Offloading

Current Scenario

- An application running on **EC2** distributes **software updates**.
- When a new update is released:
  - There is a **spike in requests**
  - The content is distributed **massively** over the network
  - This results in **high cost and CPU usage**

Goal

- **Optimize cost and performance** without changing the existing EC2 application

Strategy

- **Offload static content delivery** to:
  - **Amazon S3**: to host software update files
  - **Amazon CloudFront**: to cache and distribute the content globally
- This reduces load on EC2, improves delivery performance, and **minimizes operational costs**

Our Application: Current State

Architecture Overview

- The application is deployed across **multiple Availability Zones (AZs)**:
  - **AZ 1**, **AZ 2**, **AZ 3**
- An **Auto Scaling Group (ASG)** is used to:
  - Automatically manage the number of EC2 instances
  - Ensure high availability and scalability
- **Amazon Elastic File System (EFS)** is shared across all AZs:
  - Provides a **distributed file system**
  - Enables persistent and shared storage accessible by all EC2 instances

This setup ensures high availability and scalable compute with centralized, resilient storage.

Easy Way to Fix Things!

Updated Architecture

- The application continues to run across **three Availability Zones (AZ 1–3)** using an **Auto Scaling Group** for elasticity and high availability.
- **Amazon Elastic File System (EFS)** is still used to provide shared storage across EC2 instances.

Optimization: Amazon CloudFront

- Introduced **Amazon CloudFront** to cache and distribute content globally.

Benefits

- Reduces **load** on EC2 instances by offloading content delivery
- Improves **performance** for end users
- Minimizes **network and compute costs**

This is a minimal-impact change that significantly optimizes performance and cost without altering the core application.

Why CloudFront?

Key Advantages

- **No changes required** to the existing application architecture
- **CloudFront caches software update files at edge locations**
  - These files are **static** and do not change, making them ideal for caching

Cost and Performance Benefits

- EC2 instances are **not serverless**, but:
  - **CloudFront is serverless** and scales automatically
  - Reduces pressure on the **Auto Scaling Group (ASG)**
- **Significant savings** in:
  - EC2 instance usage

    - **Network bandwidth**
    - **High availability costs**

Summary

- CloudFront provides an **easy, low-impact optimization**
- Greatly improves **scalability** and **cost-efficiency** for existing applications distributing static content