

AWS API Gateway

Before we jump into the AWS API Gateway, we need to refresh our memory about two terms Gateway and API – what are those meant for us.

API (Application Programming Interface)

An API is a set of rules and protocols that allows different software applications to communicate with each other. It defines the methods and data formats that applications can use to request and exchange information.

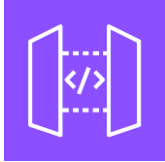
Gateway

A gateway is a server that acts as an intermediary, managing traffic between different networks or systems. In the context of an API Gateway, it handles incoming API requests, directing them to the appropriate backend services and returning the responses to the client.

🔒 API Gateway is needed to manage, secure, and optimize API requests and responses, providing a single-entry point to multiple backend services. It enhances security, scalability, and efficiency in application communication.

Now we know the API Gateway meaning, we will look into – How AWS API Gateway Operates and helps us in building our applications.

Now we know the API, what it is – we need to know who decides the structure of the API or Building Blocks of the API, those are the Resource and Method in the AWS API Gateway



AWS API Gateway

Resource

A resource represents an endpoint in your API. It is a URL path that corresponds to an object or a collection of objects.

Example: `/users`, `/orders/{orderId}`.

Resource Type	Purpose	Example
Path Resource	Main building blocks for API endpoints	<code>`/users`, `/orders/{orderId}`</code>
Proxy Resource	Captures all requests to a particular path and subpaths	<code>`/proxy/{proxy+}`</code>
Custom Domain Names	User-friendly URLs for API endpoints	<code>`api.example.com`</code>
Stage Variables	Different behaviors for different stages of API	Development vs. Production configurations
Lambda Authorizers	Custom authorization logic	Validate tokens, check user roles
Usage Plans and API Keys	Control and manage API usage	Limit requests per user per day

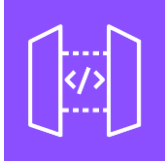
Method

A method represents an action that can be performed on a resource. It corresponds to an HTTP request method (e.g., GET, POST, PUT, DELETE).

Example:

- GET `/users` to retrieve a list of users.
- POST `/orders` to create a new order.

Method	Purpose	Use Case
GET	Retrieve information	Fetch data or resources
POST	Submit data	Create or submit data
PUT	Update or replace a resource	Modify or replace existing resources
DELETE	Remove a resource	Delete a specific resource
PATCH	Apply partial updates	Update specific fields of a resource
OPTIONS	Describe communication options for a resource	Find out allowed methods or other options
HEAD	Retrieve metadata about a resource	Check existence or metadata without data



AWS API Gateway

Together, resources and methods define the structure and behaviour of your API endpoints.

One more fundamental block, what are REST & HTTP APIs

HTTP (Hypertext Transfer Protocol) is the underlying protocol used for transferring data over the web, facilitating communication between clients and servers through methods like GET, POST, PUT, and DELETE.

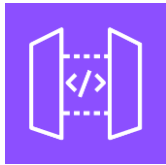
REST (Representational State Transfer) is an architectural style that leverages HTTP for creating scalable and stateless web services by organizing interactions around resources identified by URLs and using standard HTTP methods for operations.

Use HTTP:

- For general data transfer needs, such as serving web pages, images, and files.
- When implementing low-level communication between clients and servers without the constraints of a specific architectural style.

Use REST:

- When designing web services that require a clear, resource-oriented architecture.
- When you need stateless interactions, scalability, and the ability to leverage standard HTTP methods to perform CRUD operations on resources.



AWS API Gateway

How Many Types of APIs that AWS API Gateway supports

4 Types:

- HTTP API

HTTP APIs are designed for building APIs that are simpler and faster to deploy compared to REST APIs, with a focus on low latency and cost efficiency.

- WebSocket API

WebSocket APIs provide full-duplex communication channels over a single TCP connection.

- REST API

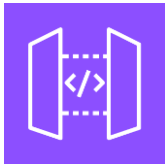
Representational State Transfer (REST) APIs are built around resources and HTTP methods.

- REST API Private

At a glance

API Type	Use Cases	Features
REST API	Web/mobile applications, microservices, third-party integrations	Caching, throttling, usage plans, AWS integrations
HTTP API	Lightweight microservices, event-driven architectures, webhooks	Lower latency, cost-efficient, simplified setup
WebSocket API	Real-time applications, streaming data, bidirectional communication	Persistent connections, real-time data transfer

PS: In REST API – AWS API Gateway supports Public and Private as well...!



AWS API Gateway

How Many Types of API End points that AWS API Gateway supports

Regional Endpoints

Regional endpoints are deployed in a specific AWS region, making them accessible within that region.

- For clients within the same region
- Could manually combine with your own CloudFront distribution for global deployment (this way you will have more control over the caching strategies and the distribution)

Edge-Optimized Endpoints

Edge-optimized endpoints use the Amazon CloudFront content delivery network (CDN) to cache API requests and responses closer to end users globally.

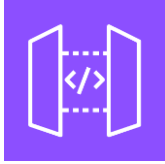
- For global clients
- Requests are routed through the CloudFront edge locations (improves latency)
- The API Gateway lives in only one region but it is accessible efficiently through edge locations

Private Endpoints

Private endpoints are accessible only within your Amazon Virtual Private Cloud (VPC) through a VPC endpoint, ensuring that the API is not exposed to the public internet.

- Can only be accessed within your VPC using an **Interface VPC endpoint** (ENI)
- Use resource policy to define access

Endpoint Type	Use Cases	Features
Regional Endpoint	Low latency for regional traffic, compliance, internal applications	Deployed in a specific region, low latency for regional users
Edge-Optimized Endpoint	Global applications, content delivery, e-commerce, media	Uses CloudFront CDN, cached responses at edge locations
Private Endpoint	Internal microservices, enterprise applications, data protection	Accessible only within VPC, ensures private network traffic



AWS API Gateway

How Many Types of API Integrations that AWS API Gateway supports

Lambda Proxy Integration

Directly integrates API Gateway with AWS Lambda functions. The request data is passed as-is to the Lambda function and the response is returned directly from the Lambda function.

Use Cases:

- When you need serverless backends.
- Handling business logic without managing servers.
- Ideal for microservices architectures.

Lambda Non-Proxy Integration

Allows more control over the request and response format. API Gateway can transform the incoming request data before passing it to the Lambda function and can transform the Lambda function's response before sending it back to the client.

Use Cases:

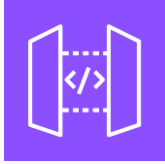
- When you need to map specific parts of the request or response to different formats.
- Useful when the input data needs significant preprocessing before being passed to Lambda.

HTTP Proxy Integration

Description: API Gateway forwards the entire request to a backend HTTP endpoint. The response from the backend is returned directly to the client.

Use Cases:

- When you have an existing HTTP backend service that you want to expose through an API.
- Quick and easy integration for existing applications.



AWS API Gateway

HTTP Custom/Non-Proxy Integration

More customizable version of HTTP proxy integration. You can map the incoming request and outgoing response to different formats.

Use Cases:


- When you need to transform the request before sending it to the backend HTTP endpoint.
- When you need to transform the response from the backend before sending it to the client.

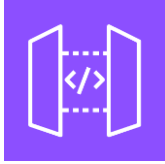
AWS Service Integration

API Gateway can directly interact with other AWS services such as S3, DynamoDB, SNS, SQS, etc. without needing a Lambda function as an intermediary.

Use Cases:

- Directly reading from or writing to DynamoDB.
- Uploading files to S3.
- Sending messages to SQS or SNS.
- Reducing the need for additional Lambda functions when direct service interactions suffice.

 This allows you to directly call other AWS services using the credentials and policies assigned to the API Gateway. You define the parameters and mappings required to call the service and process its response. There is no need for proxy/non-proxy differentiation because the integration is inherently direct and designed for specific AWS service interactions.



AWS API Gateway

Mock Integration

API Gateway returns a specified response without sending the request to a backend service.

Use Cases:

- Testing and prototyping APIs without a backend.
- Providing placeholder responses for not-yet-implemented services.

Mock integrations are used to generate API responses from API Gateway without sending the request to a backend. They are typically used for testing and development purposes. There is no concept of proxy or non-proxy here because the integration is entirely handled within API Gateway.

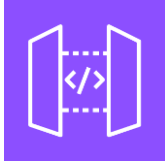
VPC Link Proxy and Non-Proxy

This resembles the Lambda Proxy and Non-Proxy, just that resources will be under a VPC and we connect to the resources only under that

Enough with theory ...! 😊

Let's Create our First API Gateway program using the Lambda Integration to print Hello World ...!, we will go to GitHub for that ..!

We will look at the URL Formats



AWS API Gateway

REST API URL format

```
https://{api-id}.execute-api.{region}.amazonaws.com/{stage-name}/{resource-name}
```

- **{api-id}**: This is the unique identifier for your API, which is generated by API Gateway.
- **{region}**: The AWS region where your API Gateway is deployed.
- **{stage-name}**: The stage you deployed your API to (e.g., prod).
- **{resource-name}**: The resource path you defined in API Gateway (e.g., hello).

HTTP API URL format

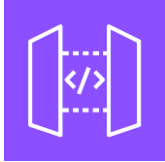
```
https://{api-id}.execute-api.{region}.amazonaws.com/{stage}/{resource-path}
```

- **{api-id}**: The unique identifier for your API.
- **{region}**: The AWS region where your API is deployed (e.g., us-east-1).
- **{stage}**: The stage of your API (e.g., dev, test, prod).
- **{resource-path}**: The specific resource path defined in your API.

WebSocket API URL Format

```
wss://{api-id}.execute-api.{region}.amazonaws.com/{stage}
```

- **{api-id}**: The unique identifier for your WebSocket API.
- **{region}**: The AWS region where your WebSocket API is deployed (e.g., us-east-1).
- **{stage}**: The stage of your WebSocket API (e.g., dev, test, prod).

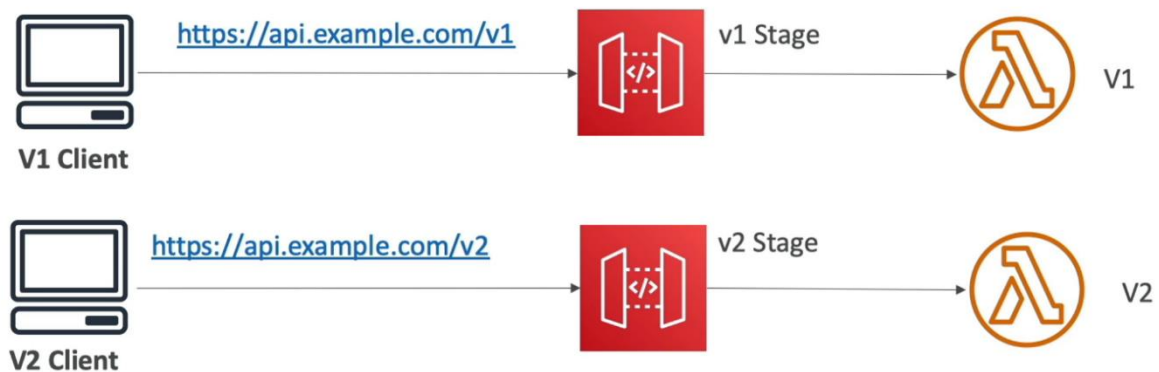


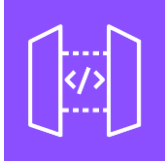
AWS API Gateway

Deployments

- Deploy the API to get the API Gateway endpoint
- If you make changes to the API, a new version is internally created. You need to deploy the API for the changes to take effect.
- Versions (changes) are deployed to stages (no limit on the number of stages) Metrics and logs are separate for each stage
- Each stage has independent configuration and can be rolled back to any version (the whole history of deployments to a stage is kept)
- Handling breaking changes using multiple stages

We want to create a new version of the application which involves breaking changes at the API level. In this case, we can deploy a new stage (v2) and build our new application there. Since multiple stages can co-exist, we can have both the versions working and once all the users have migrated to v2, we can bring down v1.





AWS API Gateway

Mapping Templates

Mapping templates in AWS API Gateway are used to transform request and response payloads between the client and backend systems. They allow you to manipulate the data format and structure to match the needs of your API and backend services. Mapping templates are written in Velocity Template Language (VTL).

In AWS API Gateway, there are primarily two types of mapping templates used for request and response transformations:

Request Mapping Templates:

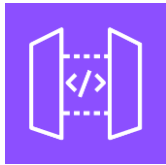
- Request mapping templates are used to transform the incoming request from the client before it reaches the backend integration.
- They allow you to modify headers, query parameters, and request body content.
- You can change the data format (e.g., JSON to XML) or structure to match the expected format of your backend service.
- Request mapping templates are applied in the "Integration Request" section of your API Gateway method configuration.

```
{
  "id": "$input.json('$.userId')",
  "name": "$input.json('$.userName')"
}
```

Response Mapping Templates:

- Response mapping templates are used to transform the response from the backend integration before it is sent back to the client.
- They allow you to modify headers, response body content, and status codes.
- You can change the response format or structure to match the expected format of your client application.
- Response mapping templates are applied in the "Integration Response" section of your API Gateway method configuration.

```
{
  "userId": "$input.json('$.id')",
  "userName": "$input.json('$.name')",
  "accountStatus": "$input.json('$.status')"
}
```



AWS API Gateway

There are 2 things, we need to know, before taking our API to the Prod ...! Those two are Stage Variables – Deployment style format

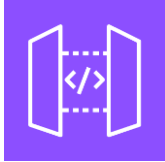
Stage Variables

- Environment variables for API gateway
- Can be changed without redeploying the API
- Stage variables are passed to context object in lambda functions
- Format to access stage variables in API gateway - `\${stageVariables.variableName}`
- Example: Stage variables to point to Lambda Aliases

Stage variables can be used to point to different Lambda aliases. Each stage points to a different lambda alias depending on the value of a stage variable. To shift traffic, we can modify the alias weights without making changes to the API gateway.

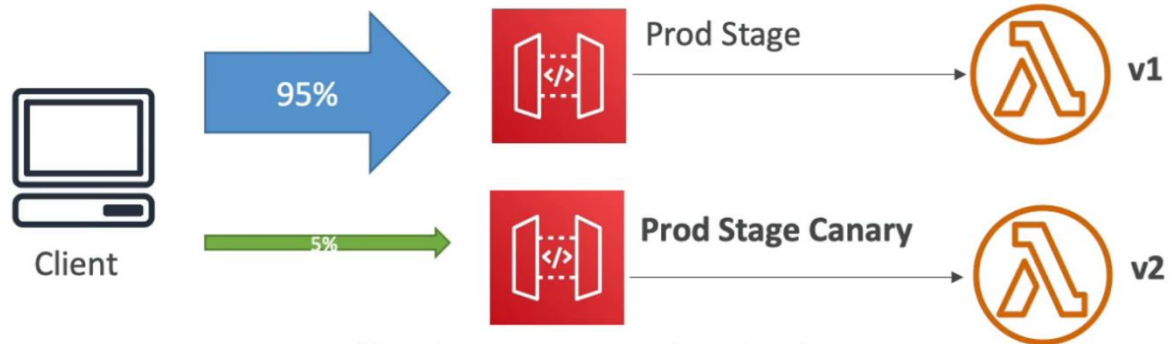
Blue Green and Canary Deployment

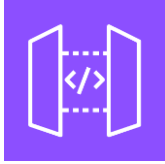
Aspect	Blue-Green Deployment	Canary Deployment
Concept	Two identical production environments (Blue & Green)	Gradual rollout to a subset of users
Process	- Deploy new version to Green	- Deploy new version to a small subset of users
	- Test on Green	- Monitor metrics and feedback
	- Switch traffic from Blue to Green	- Gradually increase user base
	- Keep Blue as backup	- Complete full rollout
Advantages	- Instant rollback	- Reduced risk
	- Zero downtime	- Flexibility
	- Simplified testing	- Cost-effective
Disadvantages	- Higher cost (maintaining two environments)	- Complex monitoring
	- Added operational complexity	- Longer deployment time
		- Complex traffic routing
When to Use	- Zero downtime needed	- Minimize risk gradually
	- Afford duplicate environments	- Robust monitoring available
	- Quick rollback desired	- Cost-effective without duplicates



AWS API Gateway

We will leverage stage variables to do the canary deployment in AWS API Gateway






AWS API Gateway

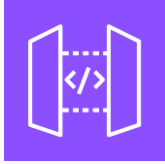
OpenAPI

AWS API Gateway OpenAPI (formerly known as Swagger) is a specification for defining RESTful APIs in a standard, language-agnostic format. **OpenAPI allows you to describe your entire API, including endpoints, operations, request/response formats, and authentication mechanisms, in a YAML or JSON file.** This specification can then be used to generate API documentation, client SDKs, and server

Using AWS API Gateway with OpenAPI:

- **Importing API Definitions:** You can import OpenAPI definitions into AWS API Gateway to create and manage RESTful APIs. This simplifies the process of setting up APIs by leveraging existing OpenAPI specifications.
- **Exporting API Definitions:** AWS API Gateway allows you to export your APIs as OpenAPI definitions. This is useful for sharing your API specifications with external developers or integrating with other tools.
- **Deployment:** With OpenAPI, you can define the entire API structure in a single file, making it easier to deploy and manage APIs across different environments (e.g., development, staging, production).

 Overall, OpenAPI is a powerful tool for designing, documenting, and managing APIs, and AWS API Gateway's support for OpenAPI makes it easier to leverage these benefits within the AWS ecosystem.



AWS API Gateway

AWS API Gateway Cache

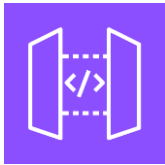
AWS API Gateway cache is a feature that allows you to store the responses of your API methods in a cache to improve the performance and reduce the latency of your API. When a request is made to an API method that has caching enabled, API Gateway checks if the response is already in the cache. If it is, API Gateway returns the cached response instead of invoking the backend integration. This can significantly reduce the load on your backend systems and improve the response times for your API clients.

Key Features of API Gateway Cache:

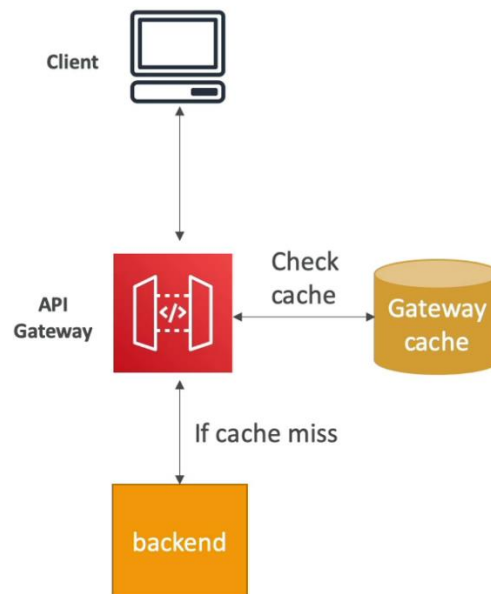
- Configurable **TTL (Time to Live)**: You can set how long the cached responses should be stored.
- **Cache Invalidation**: You can invalidate the cache manually or programmatically to ensure that stale data is not served.
- Per-Method Caching: You can enable caching on a per-method basis, allowing you to control which API methods should use the cache.
- Data Encryption: Cache data can be encrypted for security.
- Cache Capacity: You can choose the size of the cache based on your needs.

When to Use API Gateway Cache:

- **Reducing Latency**: When you need to improve the response time of your API by reducing the time it takes to fetch data from the backend.
- **Reducing Backend Load**: When you want to decrease the number of requests sent to your backend services, reducing the load and potentially lowering costs.
- **Handling High Traffic**: When your API experiences high traffic, caching can help manage the load more efficiently.
- **Consistent Data**: When the data being served by your API does not change frequently and can be cached for a certain period without becoming stale.



AWS API Gateway

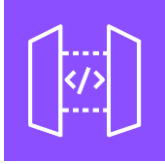


- Reduces the number of calls made to the backend
- TTL: **0 s - 1 h (default 300 sec)**
- Caching at the stage level
- Ability to override cache settings at the method level
- Cache capacity: **0.5 GB - 237 GB**
- Cache can be encrypted
- Caching is expensive (use only in production)

Cache Invalidation

Cache Invalidation is the process of removing or updating the cached data before its expiry time to ensure that the cache does not serve stale or outdated data. This is crucial to maintain data accuracy and consistency.

- Invalidate the entire cache from console
- Clients can invalidate the cache for a request by adding a header **`Cache-Control: max-age=0` in the request**. This will send the request to the backend and update the cache with the response.
- Recommended to impose an IAM policy to allow only the clients to invalidate the cache. Without an IAM policy or Authorization Disabled, anyone can invalidate the cache.
- Tick the **`Require authorization`** checkbox to only allow authorized clients to invalidate the cache.



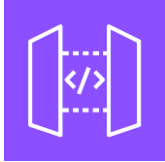
AWS API Gateway

Usage Plans

- Used to monetize the APIs
 - Which clients can access what stages and methods
 - Throttling and quota limits for each client
- Clients use API keys to access the APIs (*passed in X-API-Key header*)
- Throttling limits are applied for each API key
- Clients are billed based on the API calls using their API keys

Steps to setup a Usage Plan

- Create the API and deploy them to the right stages
- Generate API keys and distribute them to the customers
- Create a usage plan with the desired throttle and quota limits
- **Associate** API stages and **API keys with the usage plan** using **CreateUsagePlanKey API**



AWS API Gateway

Observability

Logging

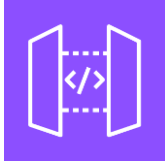
- Logs contain the request and response passing through API gateway
- **Can be enabled at the stage level**
- Can override the logging settings at the API level
- Sent to CloudWatch logs (set Log Level: ERROR, DEBUG, INFO)
- Two types of logs:
 - **Execution Logs:** log requests, responses, etc.
 - **Access Logs:** who accessed the API and how

Tracing

- Enable X-Ray to trace API calls

Metrics


- Metrics are available at the stage level, can enable detailed metrics
- Key metrics: **CacheHitCount** & **CacheMissCount**
- **Count** - request count within a period
- **IntegrationLatency** - how long the backend takes to reply to the API gateway
- **Latency** - how long client had to wait to get a response from API gateway (includes integration latency and other API Gateway overheads such as authorization)
- **4XXError** - client-side error count
 - 400: Bad request
 - 403: Access denied, WAF Filtered
 - 429: Quota exceeded, Throttle
- **5XXError** - server-side error count
 - 502: bad Gateway exception – usually for an incompatible output returned from Lambda Proxy
 - 503: Service Unavailable Exception
 - 504: Integration Failure – End point request timed out exception.

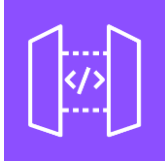


AWS API Gateway

Performance

- **Max timeout: 29 sec**
- Throttling limit - 10,000 req/sec across all APIs (account level soft limit)
 - **429 Too Many Requests** error in case of throttling
 - If one API is getting too many requests, it can throttle other APIs
 - **Set stage limits or method limits to prevent overuse**
 - Create a usage plan to throttle per customer

 *Just like Lambda Concurrency, One API that is overloaded, if not limited, can cause the Other APIs to be throttled*



AWS API Gateway

AWS API Gateway TLS Layer

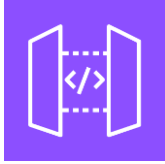
In AWS API Gateway, TLS (Transport Layer Security) certificates are used to secure the communication between clients (such as web browsers or mobile apps) and the API Gateway. By using TLS certificates, you can ensure that the data exchanged between the client and the API Gateway is encrypted and secure.

Reasons for Using TLS Certificates in API Gateway:

1. **Encryption:** TLS encrypts the data transmitted between the client and the API Gateway, protecting it from eavesdropping and man-in-the-middle attacks.
2. **Authentication:** TLS certificates help verify the identity of the API Gateway to the clients, ensuring that clients are communicating with the intended service and not an imposter.
3. **Data Integrity:** TLS ensures that the data sent between the client and the API Gateway is not tampered with during transit.

Setting Up TLS Certificates:

1. **Using ACM to Manage Certificates:**
 - **Request a Public Certificate:** You can request a public TLS certificate from AWS Certificate Manager (ACM) for your custom domain.
 - **Import a Certificate:** If you already have a TLS certificate from a third-party provider, you can import it into ACM.
2. **Associating a Certificate with a Custom Domain:**
 - **Create a Custom Domain:** In the API Gateway console, create a custom domain name and select the TLS certificate from ACM.
 - **Map API Stages to the Domain:** Map the stages of your API to the custom domain, allowing clients to access the API using the custom domain name.
3. **Enabling Mutual TLS:**
 - **Upload the CA Certificate:** In the API Gateway console, upload the certificate authority (CA) certificate that issued the client certificates.
 - **Configure the API to Require mTLS:** Enable mTLS on the API, requiring clients to present a valid client certificate issued by the CA.



AWS API Gateway

AWS API Gateway User Authentication & Authorization

Before we jump in, what is the difference between authentication and authorization we need to basically get into our head ...!

Authentication is the process of verifying the identity of a user, device, or entity. It ensures that the individual or system attempting to access a resource is indeed who they claim to be.

Authorization is the process of determining whether an authenticated entity has permission to access a specific resource or perform a specific action. It controls access rights and privileges. Common models of authorization include:

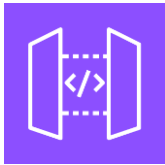
AWS API Gateway provides several types of user authentication mechanisms to secure your APIs and ensure that only authorized users can access them. The main types of user authentication in API Gateway include:

There are Majorly 03

- AWS IAM Role
- Lambda Authorization
- Cognito User Pool

On the internal side or minor aspect, we will also look at the

- API Keys
- OIDC & OAuth 2.0

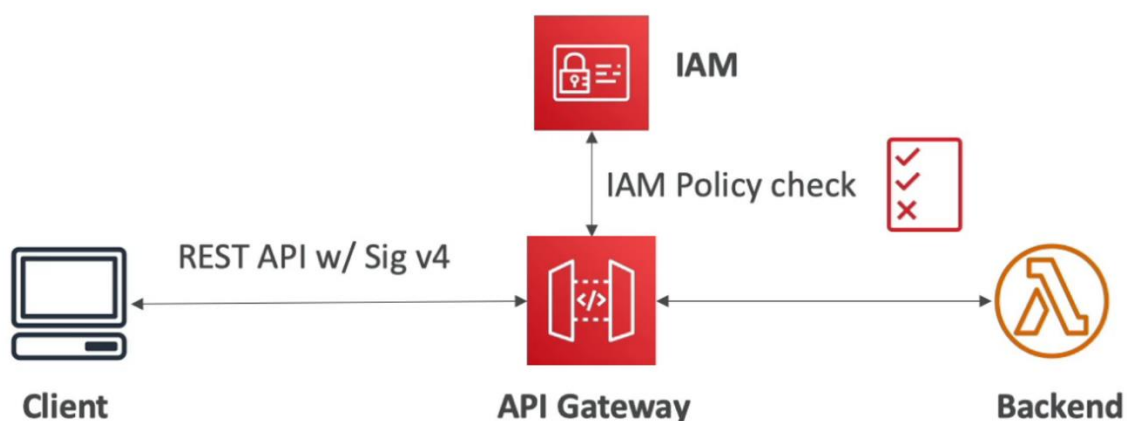


AWS API Gateway

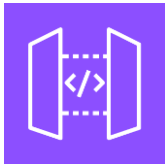
1. IAM Roles and Policies:

- **Usage:** Used to control access to your API Gateway APIs based on IAM users, groups, and roles.
- **How it Works:** You can create IAM policies that specify which API Gateway methods can be accessed by specific IAM users, groups, or roles. These policies can be attached to the relevant IAM entities.
- **Example Use Case:** An internal API accessed only by authenticated AWS users within your organization.

Authentication: **IAM** and Authorization = **IAM Policy**



- IAM User or Role for authentication
- IAM Policy applied to principals for authorization (access control)
- Fully integrated with API gateway (no custom implementation needed)
- Good to provide access within AWS
- Leverages **SigV4** to sign the credentials in the header
- **Resource-based policies** can be used to allow the following access to API gateway
 - IAM Users and Roles in another account (cross-account access)
 - IP Addresses
 - VPC Endpoint

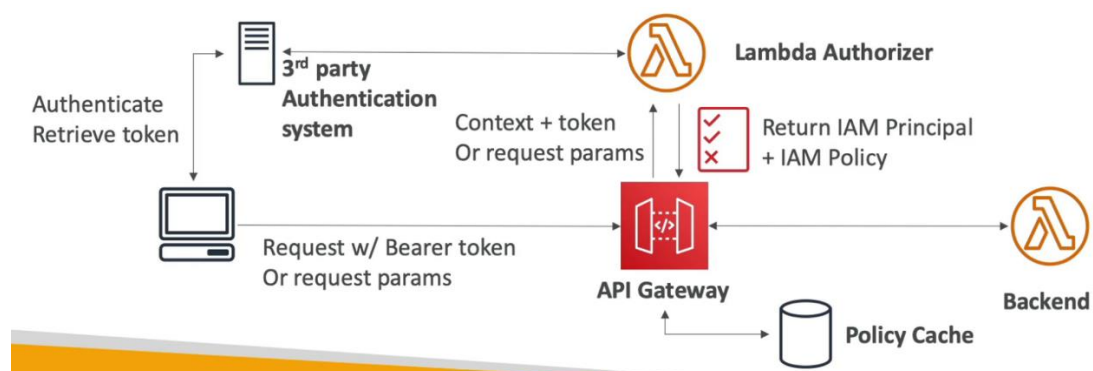


AWS API Gateway

2. Lambda Authorizers (Custom Authorizers):

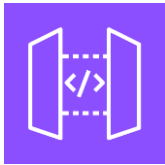
- **Usage:** Used to implement custom authorization logic using AWS Lambda functions.
- **How it Works:** When a client makes a request to your API, API Gateway invokes the specified Lambda function. The function processes the request and returns an IAM policy that allows or denies access.
- **Example Use Case:** An API that requires custom authentication logic, such as validating a JWT token against a custom user database.

Authentication: External and **Authorization = Lambda Function**



The client authenticates themselves to a 3rd party IDP and retrieve the token. The token is passed along with the request to the API gateway. The lambda authorizer takes the token, decodes it and determines the required IAM permissions. It can optionally verify the token with the IDP. The lambda authorizer then creates an IAM principal and policy granting the required permissions.

- Custom authorization logic (manual integration)
- Authentication is handled externally
- Authorization is performed by the lambda function
- Enable caching the result of authorization (recommended)
- Recommended to use this and not just rely on API keys for enhanced security.
- Two types:
 - **Token-based Lambda authorizer** (Token Authorizer) uses JWT or OAuth for 3rd party authentication system
 - **Request parameter-based Lambda authorizer** (Request Authorizer) receives the caller's identity in a combination of headers, query string parameters, stageVariables, and \$context variables.

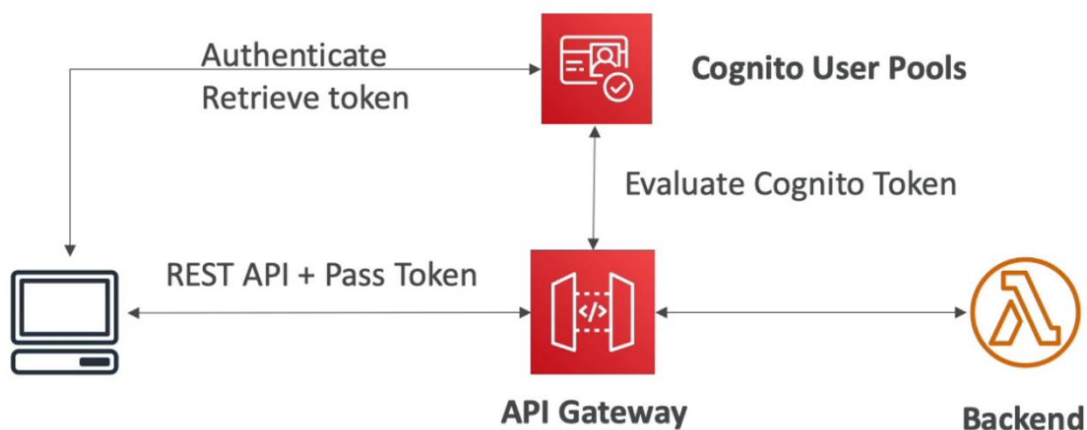


AWS API Gateway

3. Amazon Cognito User Pools:

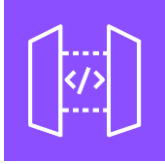
- **Usage:** Used to add user sign-up and sign-in capabilities to your API.
- **How it Works:** You can configure API Gateway to use an Amazon Cognito User Pool to authenticate and authorize users. When a client makes a request, API Gateway verifies the provided JWT token against the user pool.
- **Example Use Case:** An API for a mobile application where users need to sign up and log in to access the API.

Authentication: **Cognito User Pools** and **Authorization = API Gateway Methods**
(Must implement authorization in the back end)



The client first authenticates themselves to CUP and get the token. They pass the token along with the request to API gateway. API gateway verifies the token with Cognito before forwarding the request to the backend.

- Cognito is AWS managed identity provider. It fully manages user lifecycle and expires tokens automatically.
- Fully integrated with API gateway (no custom implementation needed)
- **Only provides authentication** (access to the API) (authorization needs to be managed in the backend)
- Good to provide access to external users



AWS API Gateway

4. API Keys:

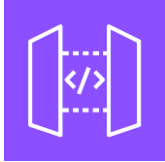
- **Usage:** Used to identify and control access for individual API clients.
- **How it Works:** You can generate API keys and associate them with usage plans. Clients must include the API key in their requests, and API Gateway checks the key against the associated usage plan.
- **Example Use Case:** An API that offers tiered access based on subscription levels, with different rate limits for each tier.

5. OpenID Connect (OIDC) and OAuth 2.0:

- **Usage:** Used to integrate with third-party identity providers for authentication and authorization.
- **How it Works:** You can configure API Gateway to use OIDC or OAuth 2.0 providers, such as Google or Facebook. API Gateway verifies the provided tokens against the identity provider.
- **Example Use Case:** An API that allows users to authenticate using their social media accounts or other third-party identity providers.

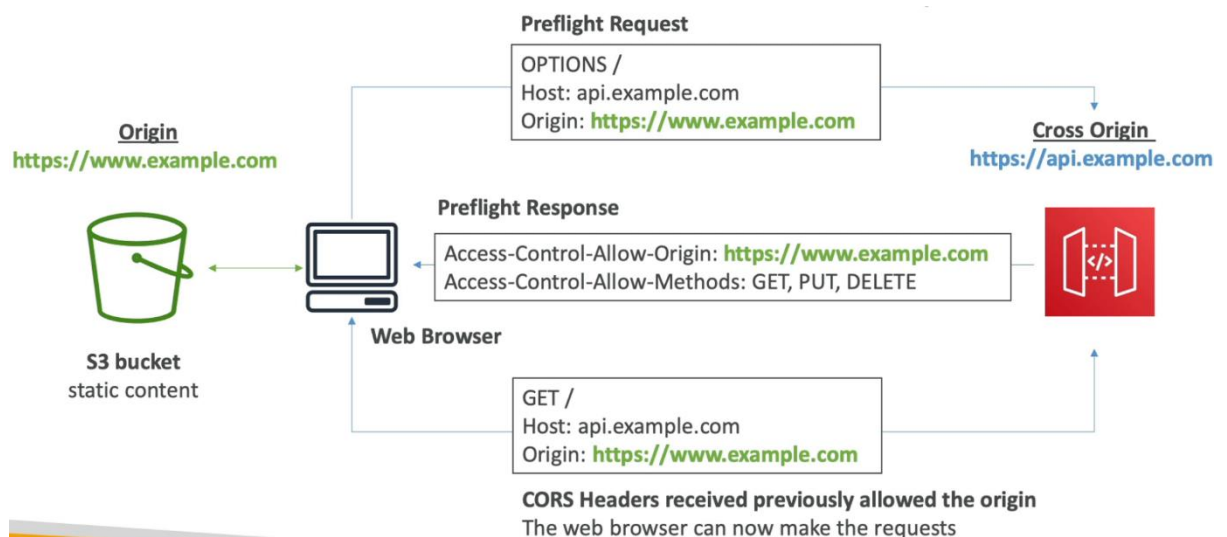
Summary of Use Cases:

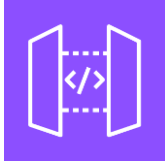
- **IAM Roles and Policies:** Secure APIs for internal use within an organization, where users are AWS IAM users.
- **Lambda Authorizers:** Implement custom authorization logic, such as validating tokens or checking custom user databases.
- **Amazon Cognito User Pools:** Add sign-up and sign-in capabilities, manage user pools, and handle user authentication for web and mobile applications.
- **API Keys:** Control access to APIs for different clients, apply usage plans, and enforce rate limits.
- **OIDC and OAuth 2.0:** Integrate with external identity providers for user authentication and authorization, allowing users to log in with their existing accounts.



Cross Origin Resource Sharing (CORS)

- Enable if you will receive API calls from another domain
- The response of pre-flight request must contain Access-Control-Allow-Origin header to allow the cross-origin client to make API calls.
- When the integration type is proxy-based, the responses are proxied to the client without modification by API gateway. So, CORS needs to be handled by the backend itself.
- For non-proxy integrations, CORS can be handled by API gateway.
- MaxAgeSeconds specifies the TTL used by browser to cache pre-flight response
- The client-side JS in the website (www.example.com) wants to make an API call to the backend hosted via API gateway at api.example.com. The browser will first make a pre-flight request to the backend asking what methods are allowed for www.example.com. If CORS is enabled, API gateway will allow the required methods.





AWS API Gateway

Numerical Questions

<https://lisireddy.medium.com/aws-api-gateway-numerical-questions-0a93638e32be>

Scenario based Questions

<https://lisireddy.medium.com/aws-api-gateway-scenario-based-questions-af7efbcfa8b8>

Use case-based Questions

<https://lisireddy.medium.com/aws-api-gateway-use-case-based-questions-f1dcc9d36a7d>