AWS Serverless : API Gateway

Example: Building a Serverless API

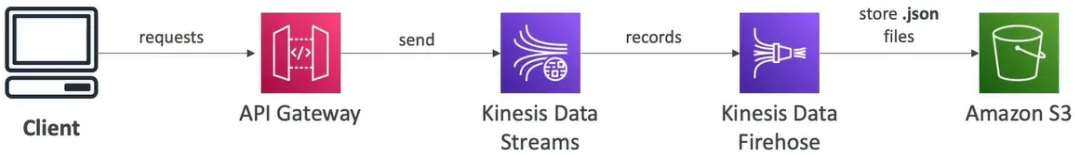
AWS API Gateway

- AWS Lambda + API Gateway : no infrastructure to manage
- support for the WebSocket Protocol
- Handle API versioning (v1,v2...)
- handle different environments (dev,test, prod,..)
- handle security (authentication [身份验证] and authorization [授权])
- create API keys, handle request throttling
- Swagger / Open API import to quickly define APIs
- Transform and validate requests and responses
- Generate SDK and API specifications
- Cache API responses

Integrations High Level

- Lambda Function
  - invoke Lambda function
  - easy way to expose REST API backed [支持的] by AWS Lambda
- HTTP
  - expose HTTP endpoints in the backend
  - example: internal [内部的] HTTP API on premise, application load balancer..
  - why ? add rate limiting, caching, user authentications, API keys,etc...
- AWS Service
  - expose any AWS API through the API Gateway
  - example : start an AWS Step Function workflow, post a message to SQS
  - why ? add authentication, deploy pulicly, rate control...

AWS Service Integration Kinesis Data Streams example



Endpoint Types

- Edge-Optimized (default): for global clients
  - requests are routed through the CloudFront Edge locations (improves latency)

    - the API Gateway still lives in only one region
- Regional:
    - for clients within the same region
    - could manually combine with CloudFront (more control over the caching strategies and the distribution)
- Private
    - can only be accessed from your VPC using an interface VPC endpoint (ENI)

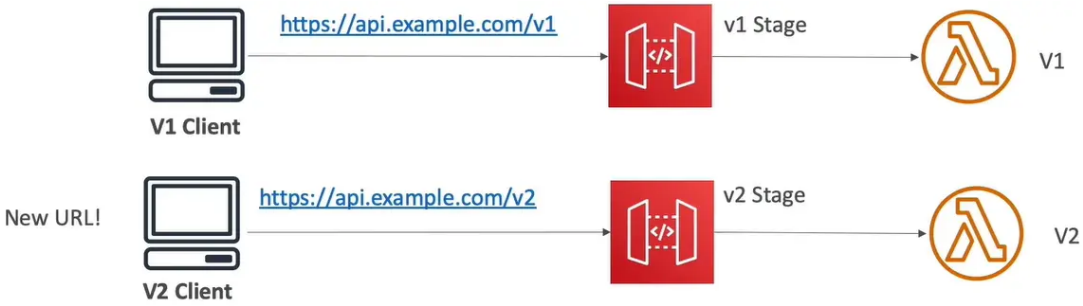
Security

- User Authentication through
    - IAM Roles (useful for internal applications)
    - Cognito (identity for external users – example mobile users)
    - Custom Authorizer (your own logic)
- Custom Domain Name HTTPS security through integration with AWS Certificate Manager (ACM)
    - if using Edge-Optimized endpoint, then the certificate must be in us-east-1
    - if using Regional endpoint , the certificate must be in the API Gateway region
    - must setup CNAME or A-alias record in Route 53

Deployment Stages

- making changes in the API Gateway does not mean they’re effective [有效的]
- you need to make a ”deployment” for them to be in effect
- it’s a common source of confusion
- changes are deployed to ”Stages” (as many as you want)
- use the naming you like for stages (dev,test,prod)
- each stage has its own configuration parameters
- Stages can be rolled back as a history of deployments is kept

Stages v1 and v2 API breaking change



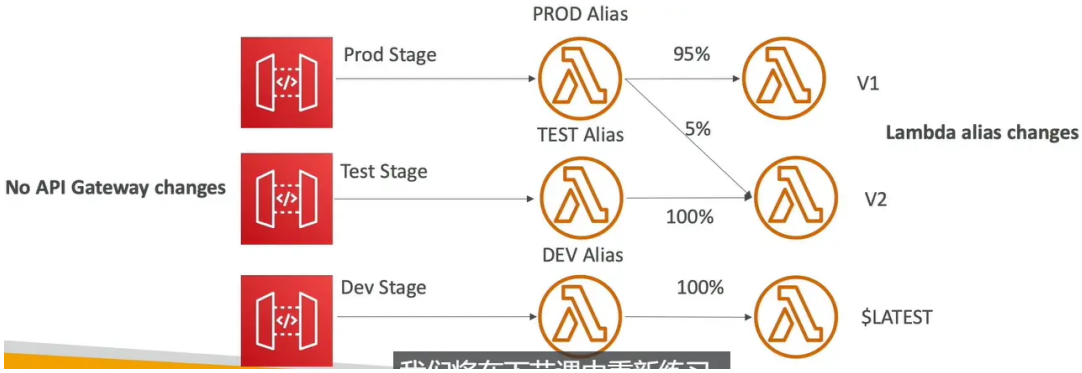
Stage Variables

- Stage variables are like environment variable for API Gateway
- use them to change often changing configuration values
- they can be used in
    - Lambda function ARN
    - HTTP Endpoint
    - Parameter mapping templates
- use cases:
    - canfigure HTTP endpoints your stages talk to (dev,test,prod..)

    - pass configuration parameters to AWS Lambda through mapping templates
- stage variables are passed to[被传递到] the ”context” object in AWS Lambda
- Format: ${stageVariables.variableName}

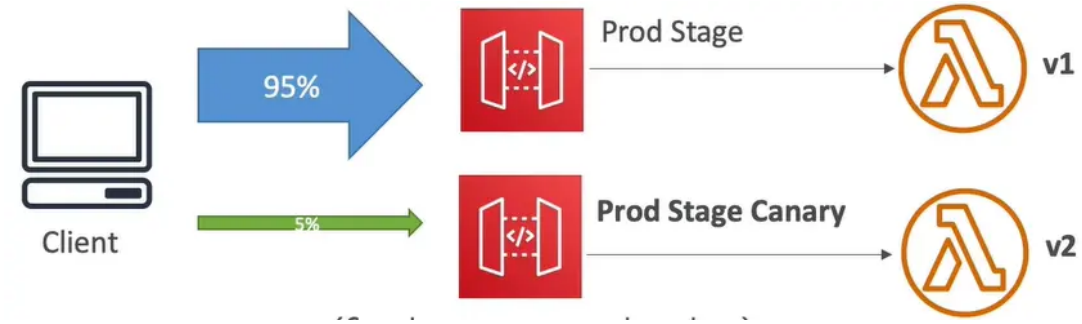
API Gateway Stage Variable & Lambda Alias

- we create a **stage variable** to indicate [指出] the corresponding [相应的] Lambda alias
- Our API gateway will automatically invoke the right Lambda function



Canary Deployment

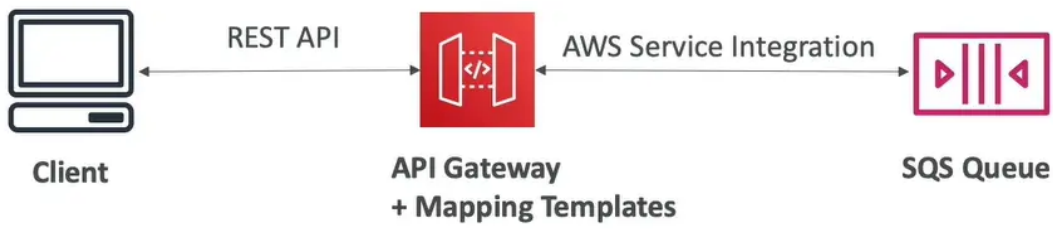
- possibility to enable canary deployments for any stage (usually prod)
- choose the % of traffic the canary channel receives



- metrics & logs are separate (for better monitoring)
- possibility to override stage variable for canary
- this is blue / green deployment with AWS Lambda & API Gateway

Integration Types

- Integration Type MOCK
    - API Gateway returns a response without sending the request to the backed
- Integration Type HTTP / AWS (Lambda & AWS Services)
    - you must configure both the integration request and integration response
    - setup data mapping using **mapping templates** for the request & response



- Integration Type AWS_PROXY (Lambda Proxy)
    - incoming request from the client is the input to Lambda
    - the function is responsible for the logic of request / response
    - no mapping template,headers, query string parameters... are passed as arguments

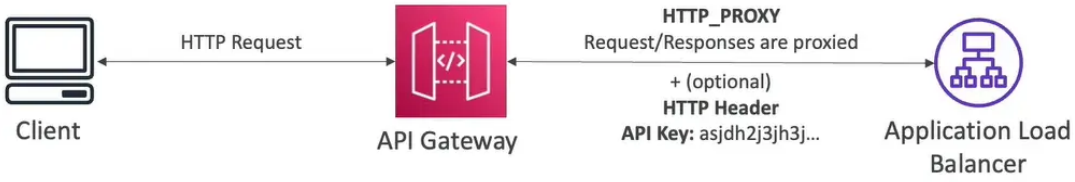
```
{
    "resource": "Resource path",
    "path": "Path parameter",
    "httpMethod": "Incoming request's method name",
    "headers": "String containing incoming request headers",
    "multiValueHeaders": "List of strings containing incomin
    "queryStringParameters": "query string parameters ",
    "multiValueQueryStringParameters": "List of query string
    "pathParameters":  "path parameters",
    "stageVariables": "Applicable stage variables",
    "requestContext": "Request context, including authorizer
    "body": "A JSON string of the request payload.",
    "isBase64Encoded": "A boolean flag"
}
```

Lambda function invocation payload

```
{
    "isBase64Encoded": "true|false",
    "statusCode": "httpStatusCode",
    "headers": { "headerName": "headerValue", ... },
    "multiValueHeaders": { "headerName": ["headerValue", "heade
    "body": "..."
}
```

Lambda function expected response

- Integration Type HTTP_PROXY
  - no mapping template
  - the HTTP request is passed to the backend
  - the HTTP response from the backend is forwarded by API Gateway
  - possibility to add HTTP Headers if need be (ex:API key)

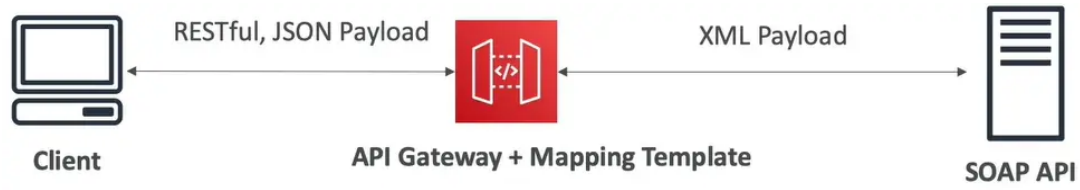


mapping Templates (AWS & HTTP Integration)

- mapping templates can be used to modify request / responses
- rename / modify **query string parameters**
- modify **body content**
- **add headers**
- uses Velocity Template Language (VTL) : for loop [循环], if etc...
- filter output results (remove unnecessary data)
- Content-Type can be set to **application/json** or **application/xml**

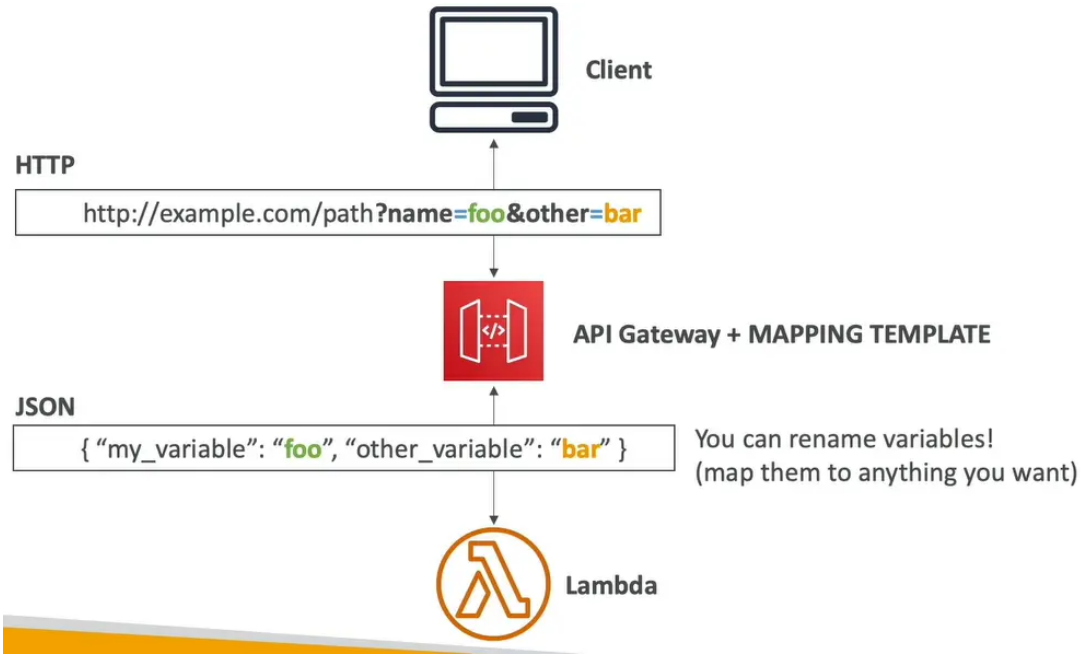
mapping example : JSON to XML with SOAP

- SOAP API are XML based,whereas REST API are JSON based



- in this case ,API Gateway should
  - extract [提取] data from the request: either path,payload or header
  - build SOAP message based on request data (mapping template)
  - call SOAP service and receive XML response
  - transform XML response to desired format (like JSON),and respond to the user

mapping example: Query String parameters



Open API spec

- common way of defining REST APIs, using API definition as code
- import existing OpenAPI 3.0 spec [规范] to API Gateway
  - method
  - method request
  - integration request
  - method response
  - + AWS extensions for API gateway and setup every single option
- can export current API as OpenAPI spec
- OpenAPI specs can be written in YAML or JSON
- Using OpenAPI we can generate SDK for our application

REST API – Request Validation

- you can configure API Gateway to perform basic validation of an API request before proceeding [继续] with the integration request
- when the validation fails, API Gateway immediately fails the request
  - returns a 400–error response to the caller
- this reduces unecessary calls to the backend

- checks:
  - the required request parameters in the URI, query string, and headers of an incoming request are included and non-blank
  - the applicable [适用的] request payload adheres to the configured JSON Schema request model of the mothod

REST API – RequestValidation – OpenAPI

- setup request validation by importing OpenAPI definitions file

```
{
    "openapi": "3.0.0",
    "info": {
        "title": "ReqValidation Sample",
        "version": "1.0.0"
    },
    "servers": [ ... ],
    "x-amazon-apigateway-request-validators": {
        "all": {
            "validateRequestBody": true,
            "validateRequestParameters": true
        },
        "params-only": {
            "validateRequestBody": false,
            "validateRequestParameters": true
        }
    }
}
```

Defines the Validators

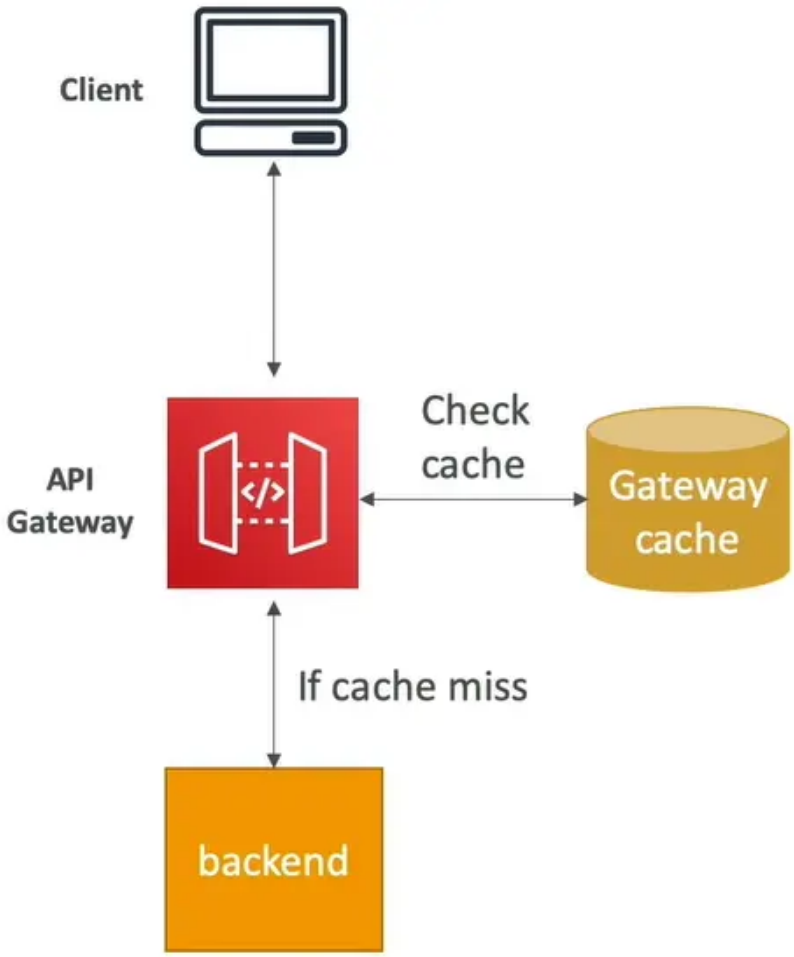
```
{
    "openapi": "3.0.0",
    "info": {
        "title": "ReqValidation Sample",
        "version": "1.0.0"
    },
    "servers": [ ... ],
    "x-amazon-apigateway-request-validator": "params-only",
    ...
}
```

Enable **params-only** Validator on all API methods

```
{
    "openapi": "3.0.0",
    "info": {
        "title": "ReqValidation Sample",
        "version": "1.0.0"
    },
    "servers": [ ... ],
    "paths": {
        "/validation": {
            "post": {
                "x-amazon-apigateway-request-validator": "all"
            }
        }
    },
    ...
```

Enable **all** Validator on the **POST /validation** method (overrides the **params-only** validator inherited from the API)

Caching API responses



- caching reduces the number of calls made to the backend
- default TTL (time to live) is 300 seconds (min:0s,max 3600s)
- caches are defined per stage
- possible to override cache settings per method
- cache encryption option
- cache capacity between 0.5GB to 237GB
- cache is expensive, make sense in production, may not make sense[合理] in dev / test

API Gateway Cache Invalidation [无效]

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "execute-api:InvalidateCache"
            ],
            "Resource": [
                "arn:...:api-id/stage-name/GET/resource-path-specifier"
            ]
        }
    ]
}
```

- able to flush [刷新] the entire cache (invalidate it) immediately

- clients can invalidate the cache with header: Cache-Control:max-age=0(with proper [正确的] IAM authorization)
- if you don't impose [强制] an InvalidateCache policy(or choose the Require authorization check box in the console),any client can invalidate the API cache

Usage Plans & API Keys

- if you want to make an API available as an offering ($) to your customers
- Usage Plan:
  - who can access one or more deployed API stages and methods
  - how much and how fast they can access them
  - uses API keys to identify API clients and meter access
  - configure throttling limits and quota [配额] limits that are enforced on individual client
- API keys
  - alphanumeric [字母数字] string values to distribute [分发] to your customers
  - Ex： WBjHxNtoAb4WPKBC7cGm64CBiblb24b4it8j]Ho9
  - can use with usage plans to control access
  - throttling limits are applied to the API keys
  - quotas lilmits is the overall number of maximum requests

Correct Order for API keys

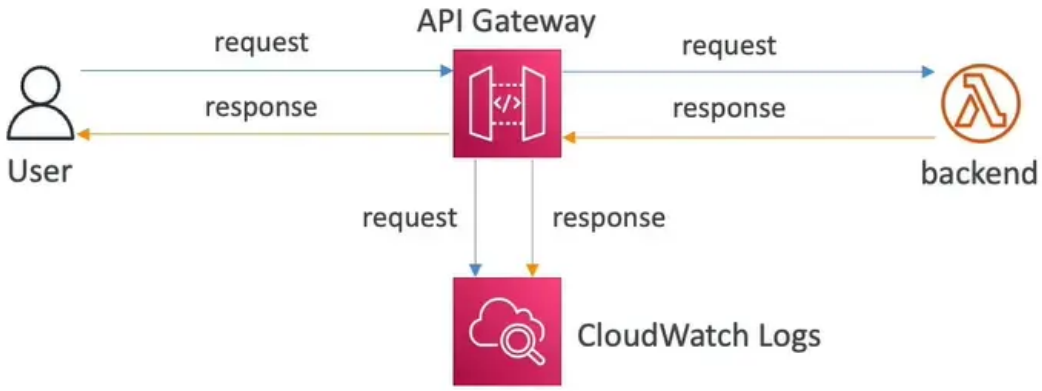
- to configure a usage plan

1. create one or more APIs,configure the methods to require an API key,and deploy the APIs to stages.
2. generate or import API keys to distribute to application developers (your customers) who will be using your API
3. create the usage plan with the desired throttle and quota limits
4. Associate API stages and API keys with the usage plan

- Callers of the API must supply [提供] an assigned API key in the x-api-key header in requests to the API.

Logging & Tracing

- CloudWatch Logs
  - log contains information about request / response body
  - enable CloudWatch logging at the stage leve (with log level – ERROR,DEBUG,INFO)
  - can override settings on a per API basis



- X-Ray
  - enable tracing to get extra information about requests in API Gateway
  - X-Ray API Gateway + AWS Lambda gives you the full pricure

CloudWatch Metrics

- metris are by stage, possibility to enable detailed metrics
- CacheHitCount & CacheMissCount : efficiency of the cache
- Count: the total number API requests in a given period
- IntegrationLatency : the time between when API Gateway relays [传达] a request to the backend and when it receives a response from the backend
- Latency： the time between when API Gateway receives a request from a client and when it returns a response to the client, the latency includes the integration latency and other API Gateway overhead.
- 4XXError (client-side) & 5XXError (server-side)

API Gateway Throttling

- Account Limit
  - API Gateway throttles requests at 10000 rps across all API
  - soft limit that can be increased upon [根据] request
- in case of throttling = > 429 too many requests (retriable error)
- can set Stage limit & Method limits to improve performance
- or you can define Usage Plans to throttle per customer

- just like Lambda Concurrency, one API that is overloaded. if not limited can cause the other APIs to be throttled

API Gateway – Errors

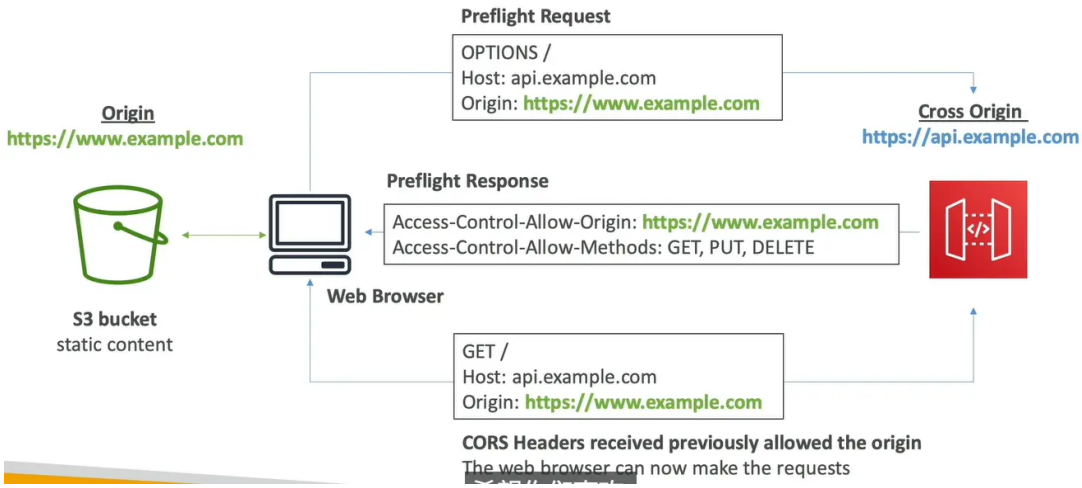
- 4xx means Client errors
  - 400: bad request
  - 403: access denied, WAT filtered
  - 429: Quota exceeded,throttle
- 5xx means Server errors
  - 502: bad Gateway exception, usually for an incompatible [不兼容] output returned from a lambda proxy integration backend and occasionally for out-of-order invocations due to heavy loads
  - 503： service unavailable exception

- 504: integration failure – ex Endpoint Request Timed-out Exception API Gateway requests time out after 29 second maximum

CORS

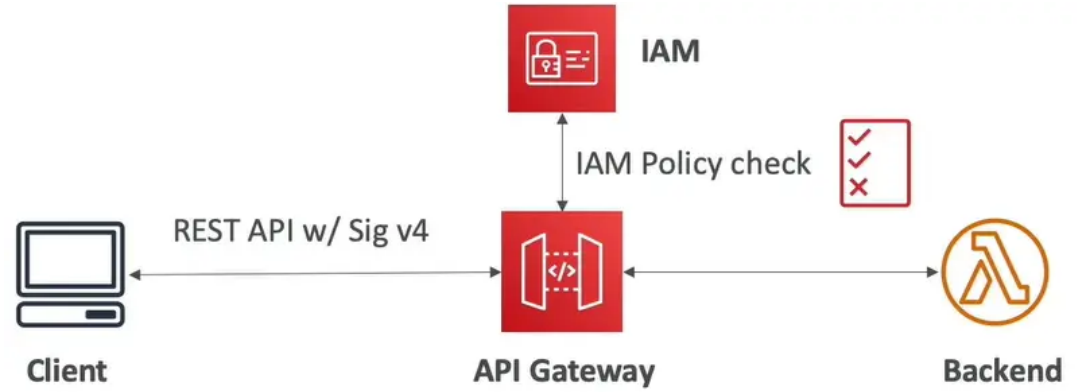
- CORS must be enabled when you receive API calls from another domain
- the OPTIONS per-flight request must contain the following headers
  - Access-Control-Allow-Methods
  - Access-Control-Allow-Headers
  - Access-Control-Allow-Origin
- CORS can be enabled through the console

CORS – Enabled on the API Gateway



Security – IAM Permissions

- create an IAM policy authorization and attach to User/ Role
- Authentication = IAM | Authorization = IAM Policy
- good to provide access within AWS (EC2,Lambda,IAM users..)
- leverage "Sig v4" capability where IAM credential are in headers



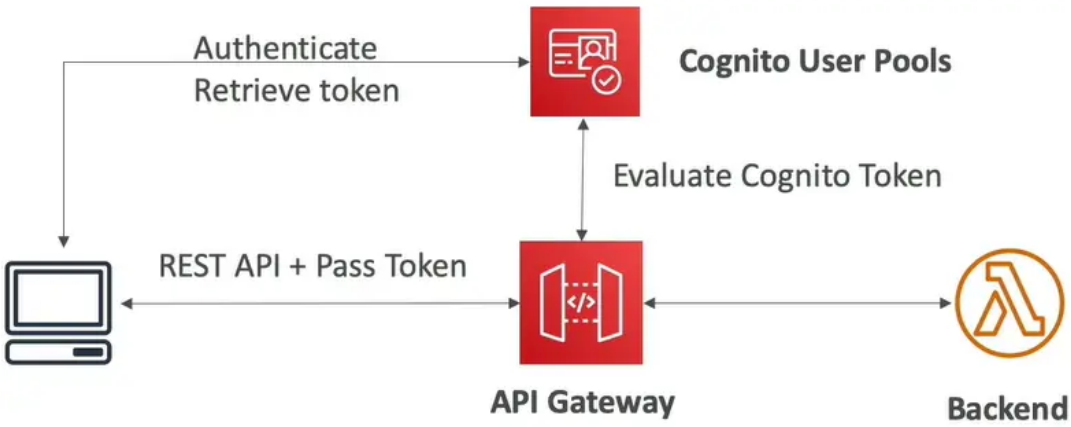
Resource Policies

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws:iam::account-id-2:user/Alice",
                    "account-id-2"
                ]
            },
            "Action": "execute-api:Invoke",
            "Resource": [
                "arn:aws:execute-api:region:account-id-1:api-id/stage/GET/pets"
            ]
        }
    ]
}
```

- Resource policies (similar to Lambda Resource Policy)
- Allow for Cross Account Access (combined with IAM Security)
- allow for a specific source IP address
- allow for a VPC Endpoint

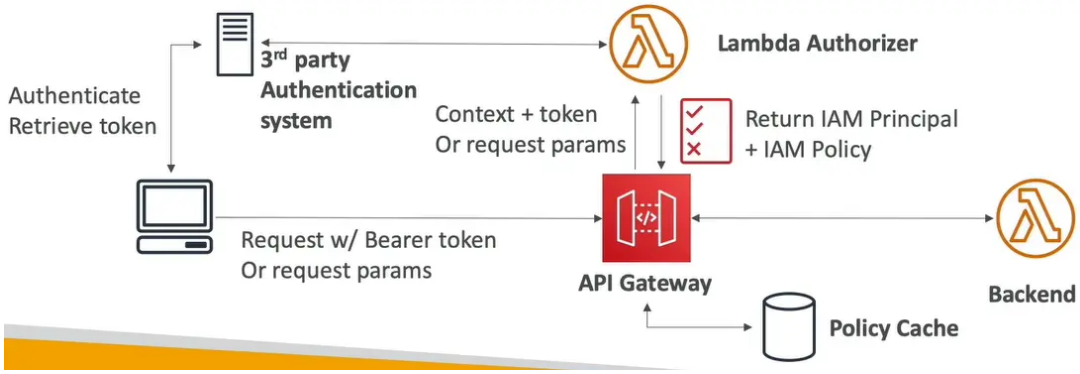
Security – Cognito User Pools

- cognito fully manages user lifecycle, token expires automatically
- API gateway verifies identity automatically from AWS cognito
- no custom implementation required
- Authentication = Cognito User Pools | Authorization = API Gateway Methods



Security – Lambda Authorizer (formerly Custom Authorizers)

- token-based authorizer (bearer [不记名] token) – ex JWT(Json Web Token) or Oauth
- a request parameter-based lambda authorizer (header,query string, stage var)
- lambda must return an IAM policy for the user, result policy is cached
- Authentication = External | Authorization = Lambda function



Security – Summary

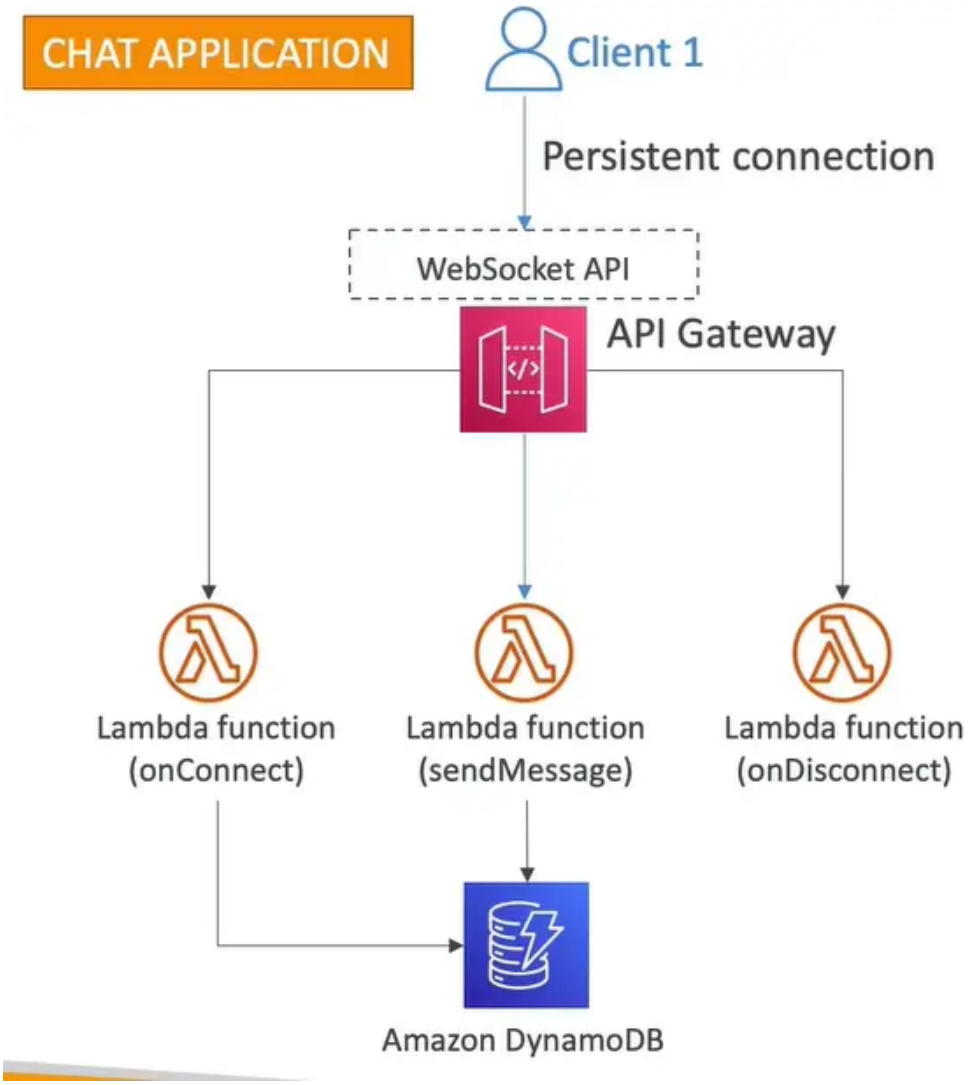
- IAM
  - great for users / roles already within your AWS account + resource policy for cross accont
  - handle authentication + authorization
  - leverage Signature v4
- Custom Authorizer
  - great for 3rd party tokens
  - very flexible in terms of [按照] what IAM policy is returned
  - handle authentication verification + authorization in the lambda function
  - pay per lambda invocation ,results are cached
- cognito user pool
  - you manage your own user pool (can be backed by facebook, google login, etc...)
  - no need to write any custom code
  - must implement authorization in the backend

HTTP API vs REST API

Authorizers	HTTP API	REST API
AWS Lambda	✓	✓
IAM	✓	✓
Resource Policies		✓
Amazon Cognito	✓ *	✓
Native OpenID Connect / OAuth 2.0 / JWT	✓	

- HTTP APIs
  - low-latency,cost-effective AWS Lambda proxy, HTTP proxy APIs and private integration (no data mapping)
  - support OIDC and OAuth 2.0 authorization, and built-in [内置] support for CORS
  - no usage plans and API keys
- REST APIs
  - all features (except Native OpenID Connect / OAuth 2.0)

WebSocket API – Overview



- what's WebSocket ?
  - two-way interactive [交互] communication between a user's browser and a server
  - server can push information to the client
  - the enable stateful [有状态的] application use cases
- WebSocket APIs are often used in real-time application such as chat applications, collaboration platfroms, multiplayer games, and financial trading platforms
- works with AWS Services (Lambda, DynamoDB) or HTTP endpoints

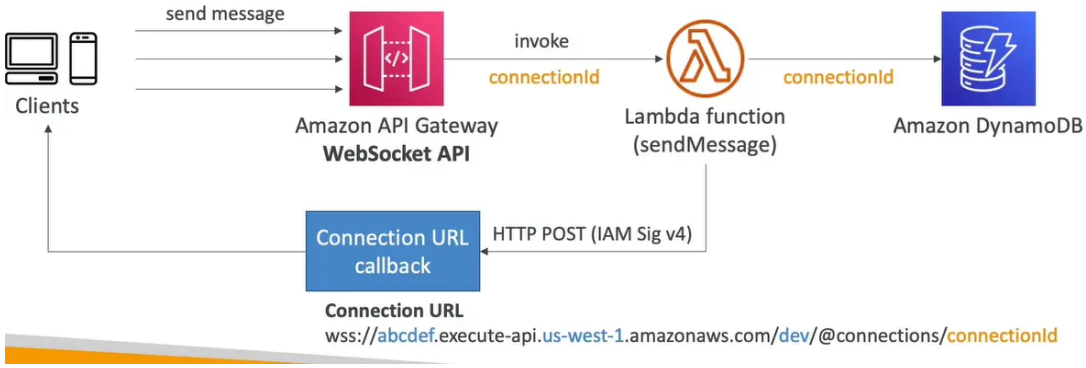
Conncting to the API

WebSocket URL
wss://[some-uniqueid].execute-api.[region].amazonaws.com/[stage-name]

Client to Server Message ConnectionID is re-used

Server to Client Message

WebSocket URL
wss://abcdef.execute-api.us-west-1.amazonaws.com/dev

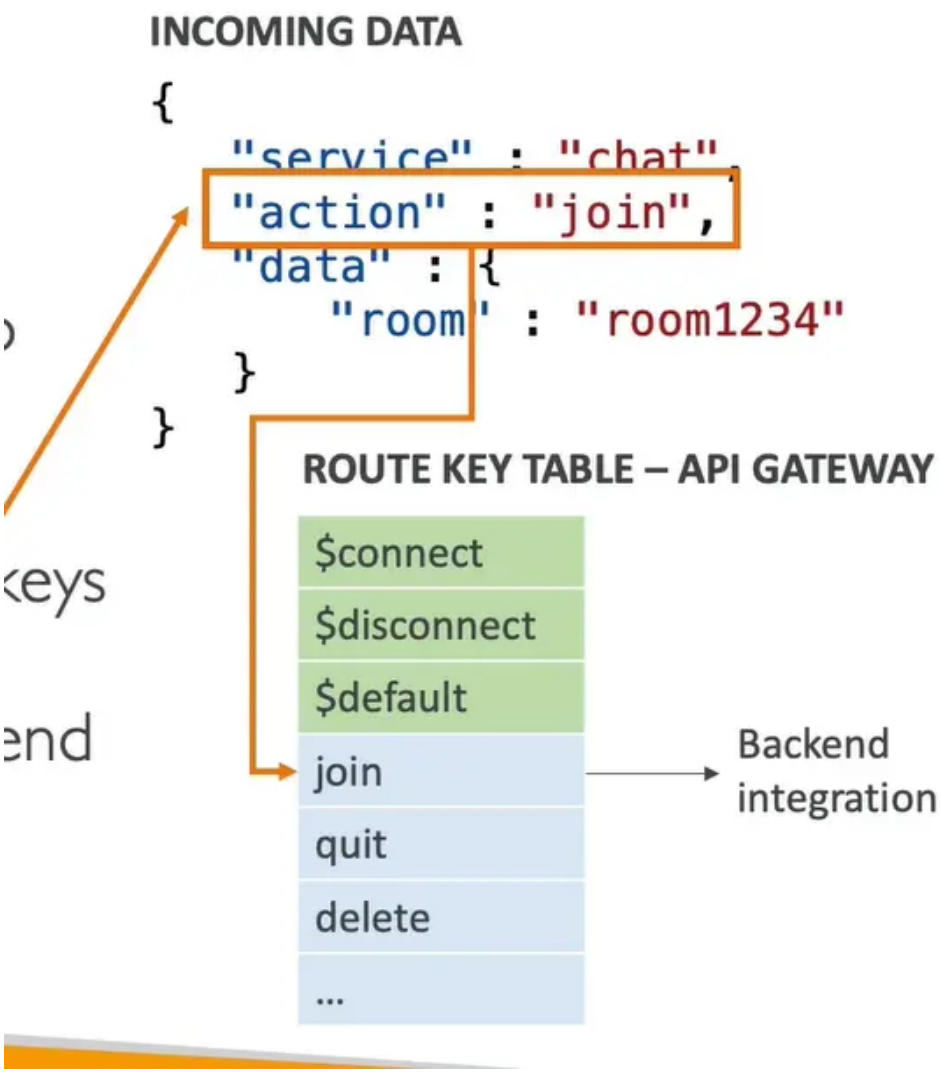


Connection URL Operations

Connection URL
wss://abcdef.execute-api.us-west-1.amazonaws.com/dev/@connections/connectionId

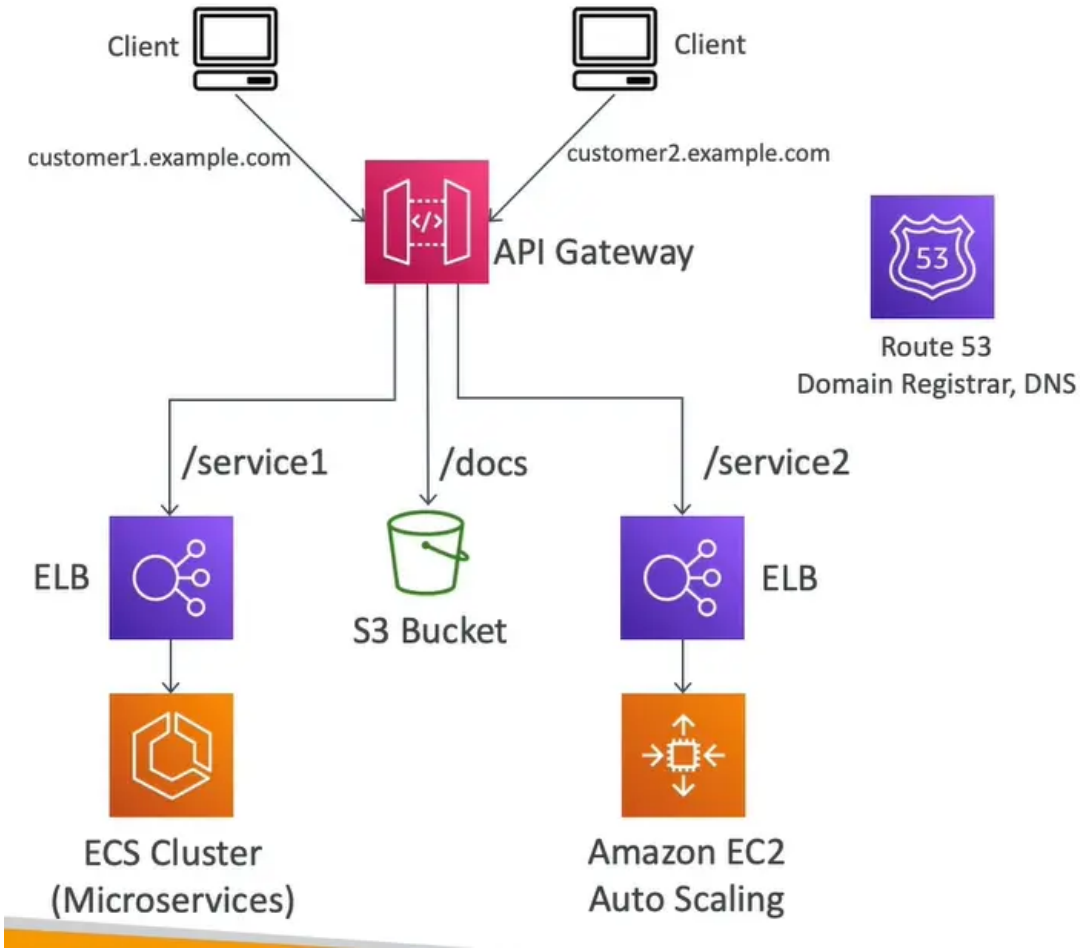
Operation	Action
POST	Sends a message from the Server to the connected WS Client
GET	Gets the latest connection status of the connected WS Client
DELETE	Disconnect the connected Client from the WS connection

WebSocket API –Routing



- incoming JSON messages are routed to different backend
- if no route = > sent to $default
- you request a *route selection expression* to selection the field on JSON to route from
- sample expression : $request.body.action
- the result is evaluated [评估] against the route keys available in your API Gateway
- the route is then connected to the backed you've setup through API Gateway

Architecture



- create a single interface for all the microservices in your company
- use API endpoints with various resources
- apply a simple domain name and SSL certificates
- can apply forwarding [转发] and transformation rules at the API Gateway level