MODULE 6 · STUDY NOTES

Developing REST APIs

Building APIs with Amazon API Gateway — REST vs. HTTP APIs, resources and methods, backend integrations, stages and deployments, access control, monitoring, and optimization

Product information for the café is now stored in a database, and Sofía wants an API to connect the website to that DynamoDB data. Before she wires up the real integration, she wants to test the front end and get Frank and Martha's approval — so she uses a mock endpoint in API Gateway to quickly build, test, and demo the API, then swaps in the real backend once it is approved. That workflow is this module in miniature. You will learn what an API is and what makes one RESTful, then meet Amazon API Gateway — a fully managed, serverless proxy and "front door" that lets you create, publish, maintain, monitor, and secure APIs at any scale. From there the module follows the life of an API: creating REST APIs with resources, methods, and routes; integrating them with Lambda and other AWS services; deploying them to stages; controlling who can call them and how often; monitoring them with CloudWatch and related tools; and optimizing them with caching and compression.

> LEARNING OBJECTIVES
>
> - Recognize APIs
> - Describe Amazon API Gateway
> - Indicate the steps for developing REST APIs with API Gateway
> - Use API Gateway to create, publish, maintain, monitor, and secure APIs

6.1 Introducing APIs

An **API** (application programming interface) is a software mechanism that simplifies development by **abstracting implementation details**, **exposing only the objects or actions a developer needs**, and **establishing how an information provider and an information user communicate**. In a typical web application, clients send requests and get responses from an API that resides on a web server, and the API provides the interface to application resources such as databases. Web services publish APIs for developers to consume — social networks such as Facebook and Twitter, and payment services such as Amazon Pay and PayPal — so developers can more easily write code that works with those applications.

> WHAT AN API DOES
>
> - **Abstracts implementation details** so callers don't need to know how the backend works.
> - **Exposes only the objects or actions** a developer actually needs.
> - **Establishes a shared contract** for how an information provider and an information user communicate.

API requests travel over a standard protocol — **SOAP** (an XML-based protocol), **WebSockets** (a bidirectional protocol), or **HTTP**. A request from a device reaches your server, and the server responds with a **status code** and possibly some data. Whether the client is a website or a native iOS or Android application, HTTP is human-readable and commonplace across the internet, which makes developing for heterogeneous systems relatively easy; even a client using SOAP usually

transmits over HTTP. WebSockets, popular for real-time and event-driven applications, use URLs that start with `ws://` instead of `http://`. With any protocol, it is a best practice to use the **secure variants** — HTTPS and `wss://` (WSS).

TABLE 6.1 HTTP response status code ranges

RANGE	MEANING	EXAMPLES
20x	Success — the request was handled	200, 201
40x	The client must do something differently — bad password, or missing information the server needed	403, 412
50x	Something went wrong that was not the client's fault — a programming error or resource failure on the server	500, 501

You often see an API categorized as either a **RESTful API** or a **WebSocket API**, but that wording is an oversimplification. **Representational State Transfer (REST)** is purely an *architectural style* — the standard way of structuring requests from a client to a server — and APIs that adhere to REST principles are called **RESTful APIs**. Non-RESTful APIs that use HTTP are perfectly valid and quite common; this module uses "RESTful" loosely, mainly to distinguish request/response APIs from WebSockets.

> **COMMON PITFALL**
>
> Not every API that uses HTTP is RESTful. REST is an *architectural style*, not a protocol, and HTTP APIs that don't follow REST principles are valid and widespread. On the exam, treat "RESTful" as a description of how requests are structured, not as a synonym for "uses HTTP."

6.2 Introducing API Gateway

An **API gateway is a proxy** that sits between your client and your server (or servers behind a load balancer). Its job is to handle the common problems developers hit when managing dynamic, server-driven applications. Without a gateway you would have to program all of that at the server, which is a poor use of server capacity — better to abstract the shared functionality out into a standalone proxy. For example, a proxy can **rate-limit** the application so the server isn't overwhelmed (like a doorman at a club), **validate required developer keys** and drop invalid requests before they reach your server (as Twitter or Google Maps do with API keys), and **limit or reroute** requests based on language headers or geographic region.

Amazon API Gateway is a fully managed, serverless service that you use to easily **create, publish, maintain, monitor, and secure APIs at any scale**. It acts as the **front door** for applications to reach data, business logic, or functionality in your backend services. It links easily to other AWS services — for example, it can intercept a client request, adjust the data if needed, forward it to AWS Lambda, and adjust Lambda's data again before sending it back — so you can go fully serverless with no server to maintain. Using API Gateway instead of letting client applications connect directly to DynamoDB gives you more control and flexibility. It supports both **RESTful and WebSocket** APIs, and you **pay only for the API calls you receive and the data transferred out**.

TABLE 6.2 API Gateway supported protocols

PROTOCOL	PATTERN	TRAITS
RESTful	Request / response using HTTP methods (GET, POST)	Short-lived communication; **stateless**
WebSockets	Two-way communication channel	Long-lived communication; **stateful**

Within RESTful APIs, API Gateway offers **two types**. A **REST API** gives the developer **full control** over requests and responses and supports features not yet available in HTTP APIs — this is *advanced mode*, where you can manipulate incoming and outgoing requests, use fine-grained custom authentication, and set up mock endpoints. An **HTTP API** simplifies APIs that need only proxy functionality and is designed for the **lowest cost and lowest latency** — savings can be up to **70%** compared to the standard REST API. (Lambda endpoints are not publicly accessible URIs, so you always need either an SDK or something like API Gateway in front to turn the request into a Lambda payload and transform the response back.) The guidance: unless you need advanced features or need to monetize your API, choose the **HTTP** type.

REST API	HTTP API
advanced mode — full control ▪ Full control over requests and responses ▪ Advanced and newer, REST-only features ▪ Mock endpoints ▪ Edge-optimized and private endpoints ▪ IAM resource policies ▪ API keys and usage plans ▪ Client SDK generation; monetization	*the default — lightweight proxy* ▪ Simple API proxy functionality ▪ Lowest cost — up to 70% savings ▪ Lowest latency ▪ Auto-deploys a default stage ▪ Regional endpoints ▪ First-class AWS service integrations

WebSockets are bidirectional and far more efficient for *chatty* applications. A typical HTTP request carries a lot of ancillary information — headers, connection bearer tokens for authentication — and that "envelope bloat" is repeated on every message even when the actual payload is tiny (a simple "Hi there"). When an application sends and receives a large volume of messages, it is better to set up a **two-way pipe** once; afterward both sides send small, efficient messages back and forth without repeating header information. WebSockets is the standard for that pipe and handles retries and broken connections; routing can invoke different backends (Lambda, Amazon Kinesis, or an HTTP endpoint) based on message content. Use **Secure WebSockets (WSS)** for the encrypted version.

TABLE 6.3 Ways to manage API Gateway APIs

PURPOSE	TOOLS
Develop	AWS Management Console · AWS CLI · OpenAPI Specification (Swagger) to document and share definitions
Deploy (infrastructure as code)	AWS SAM (a superset of CloudFormation with simplified serverless syntax; local testing) · AWS CloudFormation (YAML or JSON templates) · AWS CDK (model apps in programming languages; provisions via CloudFormation)

6.3 Creating a REST API

You can create an API with the **API Gateway console** or the **AWS CLI** (or by importing a definition). When you create and deploy a REST API — or the simpler, lower-cost HTTP API — API Gateway generates a URI with the base structure `https://{restapi_id}.execute-api.{region}.amazonaws.com/{stage_name}/`, where `{restapi_id}` is the API identifier, `{region}` is the AWS Region, and `{stage_name}` is the deployment stage. Creating an HTTP API integrated with a Lambda function (the CLI `create-api` with `--protocol-type HTTP --target <lambda-arn>`, or the console's **Add integration**) produces an API that uses a default catch-all route and a default stage set to auto-deploy. Creating a

basic **REST API is similar but needs more configuration** — notably, you must create at least one stage and deploy the API to it.

An API usually does more than one task, so a single default endpoint isn't enough — you use different **methods and routes** to do different things. For example, `GET /products` returns all products and `GET /products/34` returns product 34 (both route to a `readFn` function), while `PUT /products/34` replaces product 34 (routing to `updateFn`); the lab adds a `POST` route to create an item. Static content such as index.html, CSS, or JavaScript might live in an Amazon S3 bucket while specific paths reach specific API methods. A single function can serve "all" and "one" by branching on whether a `product_id` was passed — responding with a query instead of a scan.

TABLE 6.4 HTTP methods, actions, and example routes

METHOD	ACTION	EXAMPLE ROUTE → TARGET
GET	Read (all)	`GET /products` → readFn
GET	Read (one)	`GET /products/{product_id}` → readFn
PUT	Update / replace	`PUT /products/{product_id}` → updateFn
POST	Create	`POST /products` → create an item (lab)
{proxy+}	Catch-all	`GET /products/{proxy+}` → readFn absorbs any nested path

The **greedy path variable `{proxy+}`** lets a single integration absorb any nested paths included on the call. With `GET /products/{proxy+}`, a request like `GET /products/34/stuff/more/evenmore` is passed to `readFn` to parse and respond to with its own logic. This catch-all is simpler to build, but in general it is a **better practice to use more specific paths** — they are easier to secure, monitor, and evolve.

Creating a **WebSocket API** (`create-api --protocol-type WEBSOCKET`) additionally requires a **route selection expression**, such as `$request.body.action`, which tells your server code which method to use based on a property of the JSON message. WebSockets also needs three predefined route keys: **`$connect`** (called when a persistent connection is being initiated), **`$disconnect`** (called when the client or server disconnects), and **`$default`** (called when the route selection expression cannot be evaluated or no matching route is found).

You can also **import an API** from a definition. **OpenAPI** (formerly Swagger) is a declarative definition file that describes how your API works — useful documentation, especially when working with third-party developers. Build an API from a definition with the CLI `import-api --body file://api-definition.json`, and export an existing API's definition with `get-export`. The course does not go deeper into authoring definition files.

> **COMMON PITFALL**
>
> As a best practice, use **different paths for different actions** and connect them to different backend integrations, rather than leaning on one greedy `{proxy+}` catch-all. Specific routing makes an API easier to authorize, throttle, monitor, and change one piece at a time.

6.4 Integrating with API Gateway

API Gateway accepts and processes API calls from **three endpoint types**. Your choice affects who can reach the API and how requests are routed and protected.

TABLE 6.5 API Gateway endpoint types

TYPE	AVAILABILITY	DESIGNED FOR
Edge-optimized	REST only	Globally distributed clients; uses API Gateway's own CloudFront distribution to cut round-trip time and adds built-in DDoS protection
Regional	REST, HTTP, WebSocket	General use and clients in the same Region; best practice to put your own CloudFront distribution in front
Private	REST only	APIs reachable only from within your VPC — internal use or private microservices

For calling an API you can generally write **HTTPS requests** to the endpoint. Alternatively, API Gateway can generate a **client-side SDK** for your defined API — for Android and iOS (Swift and Objective-C), JavaScript, Ruby, and Java. SDKs are generated **only for REST APIs**. However, the generated-SDK path is non-trivial and forces you to work with API credentials on the client side (a bad security practice), so the **recommended approach is HTTPS requests to your API Gateway endpoint** — simpler to code and more secure.

Regardless of endpoint type, you can integrate with **Lambda and other AWS services**. Because Lambda and many services are **not directly reachable via HTTP URIs**, API Gateway proxies requests to them. It can run Lambda functions in your account; connect to DynamoDB or Amazon S3; start AWS Step Functions state machines; call HTTP endpoints hosted on Elastic Beanstalk, Amazon EC2, or the public internet; and integrate directly with services such as Amazon Kinesis. REST APIs also offer a **mock endpoint** that returns default data — ideal for prototyping a front end against a reliable endpoint before the real backend exists, for delivering secure index.html pages, and (as you'll see) for handling the CORS OPTIONS call. Both REST and HTTP APIs can reach resources in a VPC through a **VPC link** and an Elastic Load Balancer.

> BACKENDS API GATEWAY CAN REACH
>
> - **Lambda functions** in your account.
> - **DynamoDB** and **Amazon S3**.
> - **Step Functions** state machines.
> - **HTTP endpoints** on Elastic Beanstalk, EC2, or the public internet.
> - **Other AWS services directly** (for example, sending data to Kinesis, SQS, or SNS).
> - **Mock endpoints** (REST only) that return default data.
> - **VPC resources** via a VPC link and an Elastic Load Balancer.

Every integration has an **integration request** and an **integration response**. A client uses the API through the **method request**; if necessary, API Gateway translates it into the **integration request** the backend accepts before forwarding it. The backend returns its result as the **integration response**, which API Gateway maps to the **method response** before replying to the client. This two-way translation is what lets the client and backend speak different data shapes.

For Lambda, API Gateway **always wraps the client request** with the metadata Lambda needs; the difference between the two integration modes is how the **response** is handled. With a **non-proxy (custom)** integration, when the function responds API Gateway wraps the response into a valid HTTP response and adds a **200** on success — and it can transform the response along the way. With a **proxy** integration, the request is still wrapped, but the response **passes through unwrapped**: your function must return the required JSON output format (for example, `isBase64Encoded`, `statusCode`, `headers`, and `body`) that the browser or HTTP client understands. Proxy is simpler to set up but pushes

response-building into your function; non-proxy is the choice when you want API Gateway to wrap or transform the response.

LAMBDA PROXY INTEGRATION	LAMBDA NON-PROXY (CUSTOM) INTEGRATION
simplest passthrough	*API Gateway wraps and transforms*
▪ Request wrapped with Lambda metadata ▪ Response passes through **unwrapped** ▪ Function returns the required output format ▪ Embed the status code (403) to signal errors ▪ Simpler to set up, but you build the response	▪ API Gateway wraps the response as valid HTTP ▪ Adds a 200 status code on success ▪ Maps function errors to standard HTTP errors ▪ Transforms with **VTL mapping templates** ▪ More setup, but more control over the result

SUCCESSFUL CALL, ERROR STATUS

A Lambda function can succeed and still need to signal a problem — for instance, returning a 403 with the message 'You are not allowed to see this content.' With a **non-proxy** integration, API Gateway would otherwise treat the successful call as a 200, so you **map the error** to a standard HTTP error response and the client sees the 403 instead. With a **proxy** integration, you place the `statusCode` (403) and message directly in the required Lambda output format.

In a non-proxy integration, API Gateway can **modify or enrich** the request or response rather than merely wrapping it, using **mapping templates written in Apache Velocity Template Language (VTL)** — you can delete, add, or edit parameters. A request template might combine `firstname` and `lastname` into a single `name`, nest `city` and `state` under an `address` element, and rename a value. On the way back, you can let Lambda return plain JSON and use VTL to add the formatting, status codes, headers, and custom errors the client needs.

A **first-class integration** connects an **HTTP API route directly to an AWS service API** (similar to how API Gateway integrates with Lambda). When a request hits API Gateway, it invokes the specified AWS service API for you — for example, sending a message to an **Amazon SQS** queue or starting a **Step Functions** state machine. Each service needs a different mapping: you map the request's **BODY, QUERYSTRING, or HEADER** values to the object the target service requires (Amazon SNS expects different keys than DynamoDB). The CLI uses an `--integration-subtype` such as `SQS-SendMessage` with `--integration-type AWS_PROXY` and `--request-parameters`.

An **HTTP proxy integration** directs a route to a resource on the internet — handy when **migrating a monolith to microservices**. Map the whole monolith with API Gateway routes that forward to its endpoints (for example, point `/services` through an HTTP proxy integration to the monolith on EC2), then remap routes one by one to new Lambda function services as you break off pieces of the monolith. For **private integrations**, API Gateway reaches VPC resources with a **VPC link** to an Application Load Balancer or Network Load Balancer, or to resources registered with **AWS Cloud Map**.

You can also **transform requests and responses for HTTP APIs** using **parameter mapping**: specify which request or response parameters to modify and how — **append, overwrite, or remove** header values, and likewise for query strings and entire paths (for example, appending a header set to `$context.requestId`). Note that you **cannot map reserved headers**.

COMMON PITFALL

Proxy vs. non-proxy is a classic exam trap. A **proxy** integration does *not* format the response for you — the response passes through, so your function must return the required output format. **Non-proxy (custom)** is where API Gateway wraps the response (adding 200 on success) and where **VTL mapping templates** transform requests and responses.

6.5 Deploying an API

After you create an API you must **deploy it to make it callable**, and every deployment is **associated with a stage**. A stage must be selected when deploying a **REST or WebSocket** API; an **HTTP** API deploys to a **default stage** on creation (and any HTTP stage can be set to auto-deploy). **Stages are snapshots** of an API: you name them as you like, they are identified by API ID and stage name, and they are **included in the URL** used to invoke the API. Developers create multiple stages to differentiate versions for different environments (such as dev and prod), to connect different stages to different backends using stage variables, and to optimize a particular deployment — for example, enabling caching or throttling per stage.

WHY CREATE MULTIPLE STAGES

- **Differentiate environments** — for example a dev stage and a prod stage of the same API.
- **Connect stages to different backends** using stage variables.
- **Optimize a deployment per stage** — enable caching, customize request throttling, or configure logging.

Stage variables are name-value pairs that differ per stage, letting developers target different resources — for example, pointing a stage at a development database rather than the production database — **without writing different code**. You can also tie different API stages to different versions of a Lambda function by using **Lambda aliases** as the stage-variable values. (Lambda aliases are covered in Module 7.)

A **canary release** deploys a new version of an API for testing while the base version stays live as the production release **on the same stage**; total API traffic is split at random between the production and canary releases by a **preconfigured ratio**. You create a canary release by adding **canary settings** to a regular deployment's stage, and API Gateway sends the configured share of traffic (for example, **10.5%**) to the canary. If testing goes well, you **increase the percentage** while continuing to monitor, and eventually **promote** the canary — which fully deploys the stage to production. It is a good way to catch issues while limiting impact to most users.

SAMPLE EXAM – DEV AND PROD, ONE LAMBDA, TWO ALIASES, LEAST CONFIG

A developer wants separate dev and prod environments for an API backed by one Lambda function that has a dev alias and a prod alias, using the **LEAST configuration**. The answer: create **one REST API** and integrate it with the function using a **stage variable in place of the alias**; deploy the API to **two stages** — dev and prod — and in each stage set a stage variable to the matching alias; access each through its **stage URL**. Creating a separate API per environment, or using a **canary release** for one environment, both require more configuration. This is the module's practice certification question (answer B).

6.6 Controlling access to a REST API

API Gateway offers layered **protections against unwanted traffic** — resource policies, client certificates, AWS WAF, CORS, and throttling — plus **authorizers** that authenticate and authorize the caller. The protections harden the endpoint; the authorizers decide who is allowed in.

With REST APIs, **IAM resource policies** control access to the API's endpoint **without** needing another service such as AWS WAF — for example, allowing users in a different AWS account to call your API, or limiting your dev stage to your office IP address range. **HTTP APIs currently cannot use resource policies this way.** To use certificates for access control, you first set up a **custom domain** for the API; custom domains use **SSL certificates managed through AWS Certificate Manager (ACM)**, support **wildcard** domains, and support multiple domains through **base path mapping**.

Client certificates let a backend confirm a request truly came from API Gateway. You generate a client-side SSL certificate with API Gateway, and the backend (for example, EC2 instances) verifies the request using a **public key**; the certificate **expires after 365 days**. You can install certificates on devices such as IoT devices so API Gateway can confirm they are valid. The process uses SSL/TLS — specifically **mutual TLS**, because both API Gateway and the device present a certificate — which is possible **only when you use a custom domain**.

TABLE 6.6 Protections against unwanted traffic

PROTECTION	WHAT IT DOES	NOTE
Resource policies	Restrict who and where can call the endpoint (accounts, IP ranges)	REST only
Client certificates / mTLS	Backend confirms the request came from API Gateway using a public key	Expires after 365 days; needs a custom domain
AWS WAF	Blocks common web exploits and matches strings/regex; blocks by origin	Managed rule sets
CORS	Controls cross-origin browser requests via a preflight OPTIONS call	Browser-enforced
Throttling	Caps request rates to resist rate-based attacks	429 when throttled

AWS WAF is a web application firewall that protects web applications or APIs from common exploits; you can deploy it on CloudFront (as part of a CDN) or on API Gateway. **Managed rules** are preconfigured, regularly updated rule sets that address common security risks. With WAF you can protect APIs from exploits such as **SQL injection and cross-site scripting (XSS)**, match a specified string or regular-expression pattern in HTTP headers, methods, query strings, URIs, and the request body, and **block requests by origin** — specified IP address ranges, a specific country or region, or specific user agents, bad bots, and content scrapers.

Cross-origin resource sharing (CORS) is a browser security feature that restricts cross-origin HTTP requests initiated from scripts running in the browser. Because the **browser** enforces CORS, it is relevant only when your clients are browsers. A request is **cross-origin** when it targets a different **domain, subdomain, port, or protocol**. If your API will receive cross-origin requests, enable CORS so you accept the requests you want and block cross-origin requests from origins you don't explicitly allow. CORS uses a **preflight request** that uses the **OPTIONS** method to discover which methods are allowed; if approved, the actual request is sent, and the preflight response has a configurable **TTL**. You specify the allowed **headers, methods, and origins**; requests that don't match generate an error. Simple, read-only requests such as `GET` generally don't require CORS support.

You can set up **API Gateway as a proxy to handle all the preflight OPTIONS work**, so you write less CORS code in Lambda. CORS can be enabled and configured on **both REST and HTTP** APIs; when you configure CORS on a **REST** API, API Gateway sets up a **mock endpoint** to handle the OPTIONS method, and the setup is simplified for HTTP APIs.

Throttling protects APIs from rate-based attacks. API Gateway APIs share a **Region-wide quota of 10,000 requests per second**, and you can also set quotas **per API, per method, and per client**. A `429` (too many requests) response indicates the request was throttled. You can set throttling rates **per stage** or **per method** (REST) or **per route** (HTTP).

A **usage plan** specifies who can access deployed stages and methods and controls the rate and number of requests a client makes; within a usage plan you set throttling limits and a **quota** per **API key** (unique strings you hand out to grant access), and you can associate throttling by method to specific clients.

> **HOW API GATEWAY APPLIES THROTTLING – MOST GRANULAR FIRST**
>
> - **1. Individual method for that client** (in a usage plan).
> - **2. Client-level limits** (the usage plan overall).
> - **3. A particular method and stage.**
> - **4. The entire stage.**
> - **5. The account quota** (10,000 req/s Region-wide) — applied last, only if nothing more granular does.

For **authorization**, API Gateway offers four authorizer options: **None (open)**, **IAM**, **JWT-based (including Amazon Cognito)**, and **Lambda**. With **IAM**, the client application uses IAM credentials (usually temporary ones via client-side SDKs), and any request that isn't a signed request returns a permissions error. With **JWT**, JSON Web Tokens are used behind OpenID Connect (OIDC) or OAuth 2.0 — **REST APIs can use Amazon Cognito** as their JWT authorizer and **HTTP APIs can use third-party providers**; API Gateway evaluates the token, determines its scope, and validates whether the user can access the resource (it can validate against a Cognito user pool). You can set a distinct authorizer per route or reuse one across routes, and a JWT-authorized API needs an identity provider (Cognito is a simple one). A **Lambda authorizer** sends the request to a Lambda function that checks your custom criteria and passes an `isAuthorized` boolean to the target function, letting your code handle rejection gracefully instead of returning an abrupt 40X error.

TABLE 6.7 Authorizer options

AUTHORIZER	HOW IT WORKS	NOTES
None	Open — no authorization	Public access
IAM	Client uses IAM credentials (often temporary via SDK)	Unsigned requests are rejected
JWT (Cognito / third-party)	Validates OIDC/OAuth 2.0 tokens and their scope	REST uses Cognito; HTTP can use third-party; per-route or shared
Lambda	A function checks custom criteria	Returns an `isAuthorized` boolean to the target

> **COMMON PITFALL**
>
> The **account 10,000 req/s quota is applied last**, not first — API Gateway checks the most granular throttle first (method-per-client → client → method/stage → stage → account). Remember, too, that several controls are **REST-only**: IAM resource policies, edge-optimized and private endpoints, and client SDK generation.

6.7 Monitoring a REST API

You monitor APIs with **Amazon CloudWatch**, which collects and processes raw data from API Gateway into readable, near-real-time metrics; these statistics are **recorded for 15 months**, so you can review historical performance. By default, API Gateway sends metric data to CloudWatch **every minute**. **CloudWatch Logs** additionally provides optional logging of errors and access information for REST APIs, configured **by stage**. Combining logs and metrics lets you log errors and monitor performance together.

TABLE 6.8 Default CloudWatch metrics for REST APIs

METRIC	MEASURES
Count	Total number of API calls in a given period
IntegrationLatency	The responsiveness of the backend
Latency	Time between API Gateway receiving a request and returning a response
HTTP 400 / 500 errors	Number of client-side and server-side errors captured in a period

API Gateway automatically creates the log groups and streams for **error logs**. For **access logs**, you first create the log group and then point a stage to it when you enable access logging — with the CLI, `create-log-group` followed by `update-stage --access-log-settings ...`. You can choose from common formats such as **common log format (CLF)** and **JSON**.

TABLE 6.9 Other monitoring and audit tools

TOOL	ANSWERS THE QUESTION
AWS X-Ray	How does an individual transaction break down, and where are problems occurring across my distributed application? Combines each service's trace data into a trace; visualizes latency, isolates outliers, and helps debug
AWS Config	Does this modification comply with our rules, and how do resources relate? Provides a normalized config snapshot and flags noncompliant resources, notifying you through Amazon SNS
AWS CloudTrail	Which resources were modified, who modified them, and when? Enabled at account creation; a searchable event history of requestor identity, time, request parameters, and response elements

6.8 Optimizing API Gateway

For REST APIs, you can enable **API Gateway caching**: on a `GET` request, API Gateway checks the cache before hitting the end target (for example, a Lambda function). A **cache hit** returns the value to the client immediately — reducing calls to your endpoint and improving latency — while a **miss** goes to the target, adds the new value to the cache, and returns it, so the next identical request can be served from the cache. Cache values have a **modifiable TTL**, and you can enable **encryption** of cached data. Note that **caching is priced hourly whether or not the cache is used.** Two default metrics track it: **CacheHitCount** (requests served from the cache) and **CacheMissCount** (requests served from the backend while caching is enabled).

Payload compression reduces the amount of data sent between API Gateway and clients, which can lower cost and improve performance. Choose a supported **content coding** (`deflate`, `gzip`, or `identity`), set a **`minimumCompressionSize`** to tell API Gateway when to compress a payload, and send requests with a compressed payload and an appropriate **`Content-Encoding`** header. By default API Gateway **decompresses the method request** payload, but you must **configure compression of the method response** payload. A `minimumCompressionSize` of **0 compresses all payloads** (set via `update-rest-api --patch-operations op=replace,path=/minimumCompressionSize,value=0`). Because compressing very small payloads can actually increase size and add latency and compute time, **test to find an optimal value.**

COMMON PITFALL

Two optimization gotchas: **caching is billed hourly whether or not it is used**, so enable it where repetitive `GET` traffic earns the cost; and a `minimumCompressionSize` of **0 compresses everything**, which can backfire on tiny payloads — run test cases before committing to a value.

Module summary

- An API abstracts implementation details and defines how an information provider and user communicate; clients send requests and get responses over protocols such as HTTP, SOAP, or WebSockets, with 20x (success), 40x (client must change something), and 50x (server-side failure) status codes.
- REST is an architectural style (Representational State Transfer) for structuring client-to-server requests; APIs that follow it are RESTful, but non-RESTful HTTP APIs are valid and common too.
- An API gateway is a proxy that offloads cross-cutting work (rate limiting, key validation, routing); Amazon API Gateway is a fully managed, serverless front door for REST, HTTP, and WebSocket APIs, billed per call and per data transferred out.
- REST API = full control and advanced, REST-only features (mock endpoints, edge-optimized/private endpoints, resource policies, API keys and usage plans, client SDKs); HTTP API = lightweight, lowest cost (up to 70% savings) and latency — default to HTTP unless you need a REST-only feature or monetization.
- Create APIs via the console, the CLI, or an OpenAPI import; map different paths and methods (GET/POST/PUT) to different backends; {proxy+} absorbs nested paths; WebSocket APIs need a route selection expression plus the $connect, $disconnect, and $default route keys.
- Endpoint types are edge-optimized (REST, CloudFront/DDoS, global), Regional (all types), and private (REST, VPC-only); integrate with Lambda, DynamoDB, S3, Step Functions, HTTP endpoints, other AWS services, mock endpoints, and VPC resources through a VPC link.
- Integrations translate a method request into an integration request and map the integration response to the method response; Lambda proxy passes the response through unwrapped (your function returns the required output format), while non-proxy (custom) wraps and transforms with VTL mapping templates.
- Deployments attach to a stage (a named snapshot in the invoke URL); stage variables point stages at different backends such as Lambda aliases without code changes; canary releases split a percentage of a stage's traffic to a new version before promotion.
- Control access with resource policies (REST only), client certificates / mutual TLS, AWS WAF, CORS (browser preflight OPTIONS), and throttling (applied most-granular-first; 10,000 req/s Region quota; 429 = throttled); authorize callers with None, IAM, JWT (Cognito or third-party), or a Lambda authorizer.
- Monitor with CloudWatch metrics (Count, IntegrationLatency, Latency, 400/500 errors; retained 15 months) and per-stage access logs, plus X-Ray, AWS Config, and CloudTrail; optimize with caching (GET, adjustable TTL, hourly pricing) and payload compression (deflate/gzip/identity, minimumCompressionSize).

Review questions

1. **What is REST, and is every API that uses HTTP RESTful?**

   Answer: REST (Representational State Transfer) is an architectural style — the standard way of structuring requests from a client to a server; an API that adheres to REST principles is RESTful. Not every HTTP API is RESTful: non-RESTful APIs that use HTTP are valid and common.

2. **Compare the REST API and HTTP API types in API Gateway, and say when to choose each.**

   Answer: The REST API type gives full control and advanced, REST-only features (mock endpoints, edge-optimized and private endpoints, resource policies, API keys and usage plans, client SDKs, monetization). The HTTP API type is a lightweight proxy with the lowest cost (up to 70% savings) and latency. Choose HTTP unless you need an advanced REST-only feature or must monetize the API.

3. **Explain the difference between a Lambda proxy and a non-proxy (custom) integration.**

   Answer: Both wrap the request with the metadata Lambda needs. Proxy passes the response through unwrapped — the function must return the required JSON output format (statusCode, headers, body). Non-proxy has API Gateway wrap the response (adding 200 on success) and can transform it with VTL mapping templates; you map function errors to standard HTTP errors so clients don't see 200.

4. **A team runs dev and prod off one Lambda function with two aliases and wants the least configuration. What do they do?**

   Answer: Create one REST API integrated with the function via a stage variable, deploy it to two stages (dev and prod), and set the stage variable in each stage to the matching alias, accessed by stage URL. Separate APIs per environment or a canary release would require more configuration.

5. **List API Gateway's protections against unwanted traffic and its authorizer options.**

   Answer: Protections: resource policies (REST only), client certificates / mutual TLS, AWS WAF, CORS, and throttling. Authorizers: None (open), IAM, JWT-based (including Amazon Cognito), and Lambda.

6. **In what order does API Gateway apply throttling, and what does a 429 mean?**

   Answer: Most granular first: the individual method for that client, then client-level limits (usage plan), then a particular method and stage, then the entire stage, and finally the account quota (10,000 requests per second Region-wide). A 429 'too many requests' means the request was throttled.

7. **How do you monitor and optimize an API Gateway REST API?**

   Answer: Monitor with CloudWatch default metrics (Count, IntegrationLatency, Latency, HTTP 400/500 errors; retained 15 months) and per-stage access logs, plus X-Ray tracing, AWS Config, and CloudTrail. Optimize with response caching (GET requests, adjustable TTL, optional encryption, priced hourly; CacheHitCount/CacheMissCount) and payload compression (deflate/gzip/identity content codings and a minimumCompressionSize).