**REST API**

**Representational State Transfer:**

It is an architectural style for providing standards between computer systems on the web, making it easier for systems to communicate with each other.

**Separation of Client and Server:**

The code on the client side can be changed at any time without affecting the operation of the server, and the code on the server side can be changed without affecting the operation of the client.

By using a REST interface, different clients hit the same REST endpoints, perform the same actions, and receive the same responses.

**Statelessness:**

Systems that follow the REST paradigm are stateless, meaning that the server does not need to know anything about what state the client is in and vice versa.

**Communication between Client and Server**

In the REST architecture, clients send requests to retrieve or modify resources, and servers send responses to these requests.

Standard ways to make requests and send responses:

**Making Requests:**

The client requests the server to retrieve or modify the data on the server. The request consists of:

1. HTTP verb - kind of operation to perform.

There are 4 basic HTTP verbs:

* 1. GET — retrieve a specific resource (by id) or a collection of resources.
  2. POST — create a new resource.
  3. PUT — update a specific resource (by id)
  4. DELETE — remove a specific resource by id.

1. Headers and Accept Parameters – allows the client to pass along information about the request.

In the header of the request, the client sends the type of content that it can receive from the server. This is called the Accept field, and it ensures that the server does not send data that cannot be understood or processed by the client. The options for types of content are **MIME Types (or Multipurpose Internet Mail Extensions).**

MIME Types, used to specify the content types in the Accept field, consist of a type and a subtype. They are separated by a slash (/).

For example, a text file containing HTML would be specified with the type text/html. If this text file contained CSS instead, it would be specified as text/css. A generic text file would be denoted as text/plain. This default value, text/plain, is not a catch-all, however. If a client is expecting text/css and receives text/plain, it will not be able to recognize the content.

Other types and commonly used subtypes:

* image - image/png, image/jpeg, image/gif
* audio - audio/wav, audio/mpeg
* video - video/mp4, video/ogg
* application - application/json, application/pdf, application/xml, application/octet-stream

For example, a client accessing a resource with id 23 in an articles resource on a server might send a GET request like this:

GET /articles/23

Accept: text/html, application/xhtml

The Accept header field, in this case, is saying that the client will accept the content in text/html or application/xhtml.

1. Path to the resource.

Requests must contain a path to a resource that the operation should be performed on. In RESTful APIs, paths should be designed to help the client know what is going on. Conventionally, the first part of the path should be the plural form of the resource. This keeps nested paths simple to read and easy to understand.

A path like fashionboutique.com/customers/223/orders/12 is clear in what it points to, even if you’ve never seen this specific path before, because it is hierarchical and descriptive. We can see that we are accessing the order with id 12 for the customer with id 223.

1. Optional message body that contains data.

**Sending Responses:**

**Content Types:**

In cases where the server is sending a data payload to the client, the server must include a content-type in the header of the response. This content-type header field alerts the client to the type of data it is sending in the response body. These content types are MIME Types, just as they are in the accept field of the request header. The content-type that the server sends back in the response should be one of the options that the client specified in the accept field of the request.

For example, when a client is accessing a resource with id 23 in an articles resource with this GET Request:

GET /articles/23 HTTP/1.1

Accept: text/html, application/xhtml

The server might send back the content with the response header:

HTTP/1.1 200 (OK)

Content-Type: text/html

This would signify that the content requested is being returned in the response body with a content-type of text/html, which the client said it would be able to accept.

**Response Codes:**

|  |  |
| --- | --- |
| **Status code** | **Meaning** |
| 200 (OK) | This is the standard response for successful HTTP requests. |
| 201 (CREATED) | This is the standard response for an HTTP request that results in an item being successfully created. |
| 204 (NO CONTENT) | This is the standard response for successful HTTP requests, where nothing is being returned in the response body. |
| 400 (BAD REQUEST) | The request cannot be processed because of bad request syntax, excessive size, or another client error. |
| 403 (FORBIDDEN) | The client does not have permission to access this resource. |
| 404 (NOT FOUND) | The resource could not be found at this time. It is possible it was deleted or does not exist yet. |
| 500 (INTERNAL SERVER ERROR) | The generic answer for an unexpected failure is if there is no more specific information available. |

For each HTTP verb, there are expected status codes a server should return upon success:

GET - return 200 (OK)

POST - return 201 (CREATED)

PUT - return 200 (OK)

DELETE - return 204 (NO CONTENT) If the operation fails, return the most specific status code possible corresponding to the problem that was encountered.

**Examples of Requests and Responses:**

Let’s say we have an application that allows you to view, create, edit, and delete customers and orders for a small clothing store hosted at fashionboutique.com. We could create an HTTP API that allows a client to perform these functions:

If we wanted to view all customers, the request would look like this:

GET http://fashionboutique.com/customers.

Accept: application/json

A possible response header would look like:

Status Code: 200 (OK)

Content type: application/json

followed by the customer's data requested in application/json format.

Create a new customer by posting the data:

POST http://fashionboutique.com/customers

Body:

{

“customer”: {

“name” = “Scylla Buss”,

“email” = “scylla.buss@codecademy.org”

}

}

The server then generates an id for that object and returns it back to the client, with a header like:

201 (CREATED)

Content-type: application/json

To view a single customer we GET it by specifying that customer’s id:

GET http://fashionboutique.com/customers/123

Accept: application/json

A possible response header would look like:

Status Code: 200 (OK)

Content-type: application/json

followed by the data for the customer resource with id 23 in application/json format.

We can update that customer by PUT ting the new data:

PUT http://fashionboutique.com/customers/123

Body:

{

“customer”: {

“name” = “Scylla Buss”,

“email” = “scyllabuss1@codecademy.com”

}

}

A possible response header would have Status Code: 200 (OK), to notify the client that the item with id 123 has been modified.

We can also DELETE that customer by specifying its id:

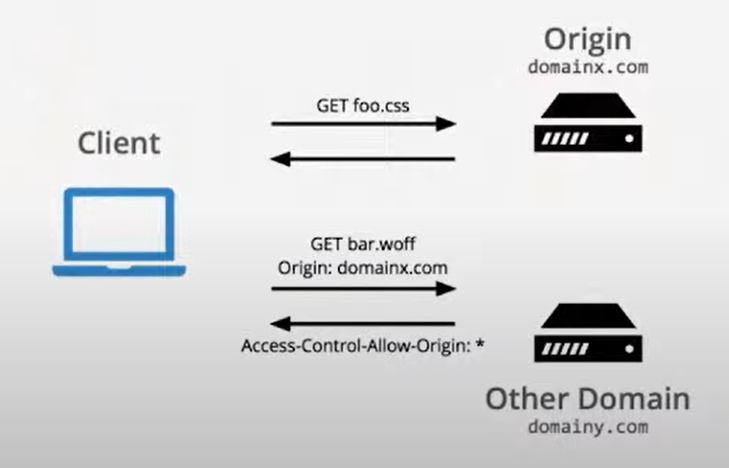
DELETE http://fashionboutique.com/customers/123

The response would have a header containing Status Code: 204 (NO CONTENT), notifying the client that the item with id 123 has been deleted, and nothing in the body.

---------------------------------------------------------------------------------------------------------------------

**CORS:**

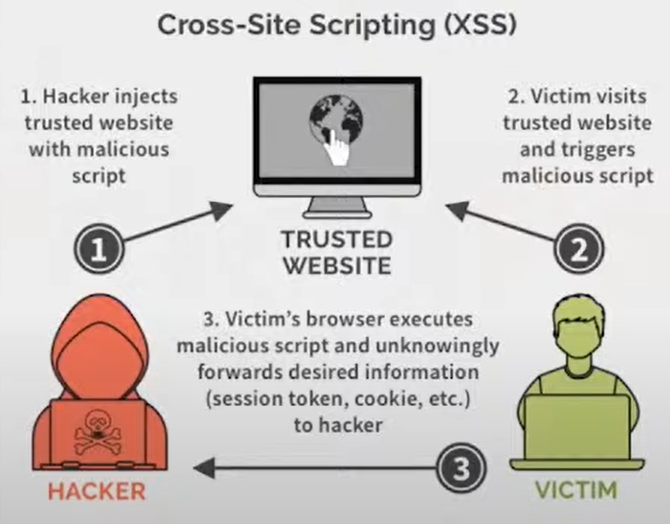
Cross-origin resource sharing (CORS) is a mechanism that **allows restricted resources** on a web page to be requested from another domain outside the domain from which the first resource was served.



CORS is a security mechanism employed by browsers to prevent the browsers from making calls to another website. A request for a resource outside of the origin is known as a cross-origin request. CORS is a specification that manages cross-origin requests.

**Need for CORS:**

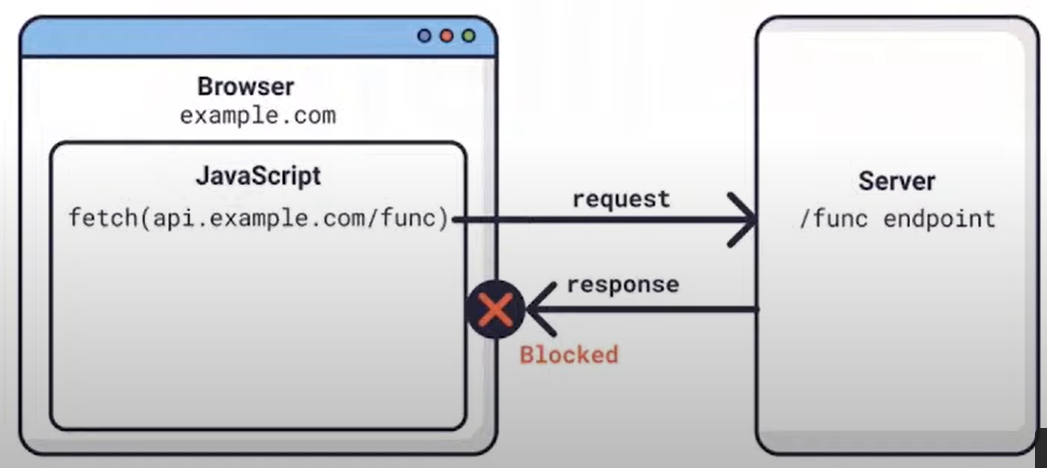
To prevent websites from tampering with one another, web browsers implement a security measure known as **same-origin policy**.

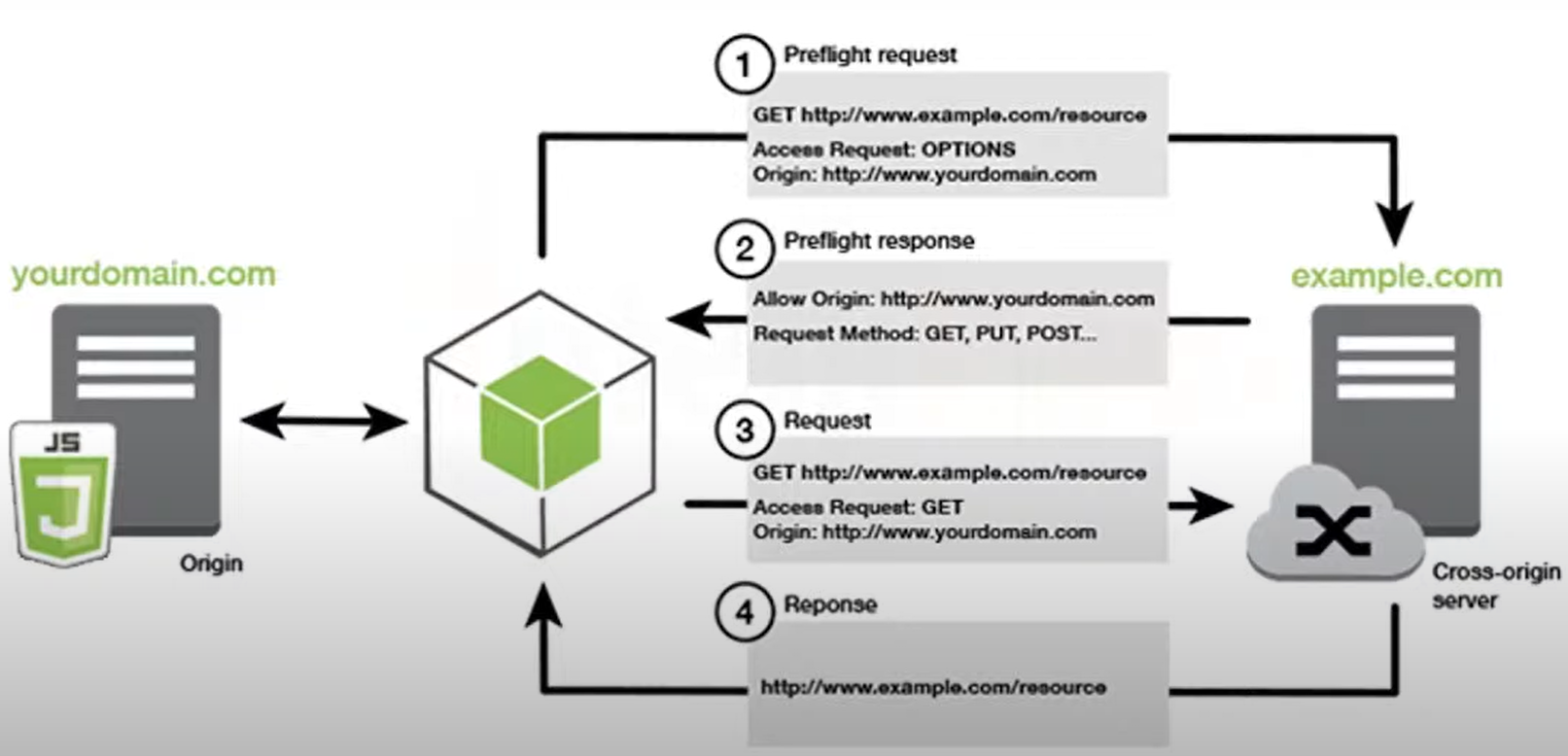


In cases where **cross-domain scripting** is desired, CORS allows web developers to work around the same-origin policy. CORS adds HTTP headers that instruct web browsers on how to use and manage cross-domain content. The browser then allows or denies access to the content based on the security configuration. That is if the HTTP header has the CORS content it is allowed inside the domain else not allowed.

**HOW does CORS work?**

When a browser executes a script that references a resource on another domain, it requests the content directly from the second domain. The second domain determines whether to serve the content by validating the first domain, which is included as a part of the request. The second domain then returns either the content or an error message back to the browser, bypassing the first domain entirely.





The user opens a resource(yourdomain.com) on a page that references another domain (example.com).

First domain – yourdomain.com, second domain – example.com

1. Preflight request: The user browser creates a connection to the second domain adding an origin HTTP header to the request which contains the first domain.
2. Preflight response: The second domain replies with an access control origin HTTP header which lists the domains allowed to make the CORS requests. The wildcard or Asterisk allows all domains to make requests.
3. Request: If the first domain is allowed to make the request the second domain responds with the requested content.

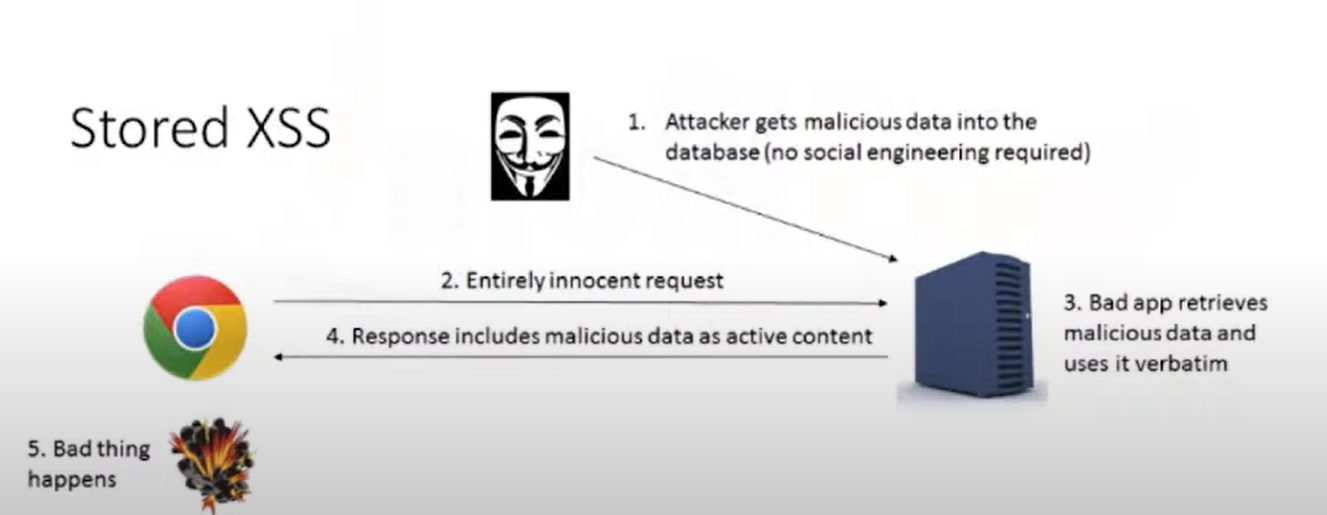
The access control allows the origin header means it is defined in the second domain server configuration so if the header doesn’t contain wildcards and the first domain is not explicitly included the browser displays an error message i.e., if CORS header is not present in the request of the second domain it will display an error message.

**Example of CORS usage:**

CORS is an essential feature of online storage services such as Amazon S3, Service providers configure S3 to allow CORS requests from their website’s domain.

**Conclusion:**

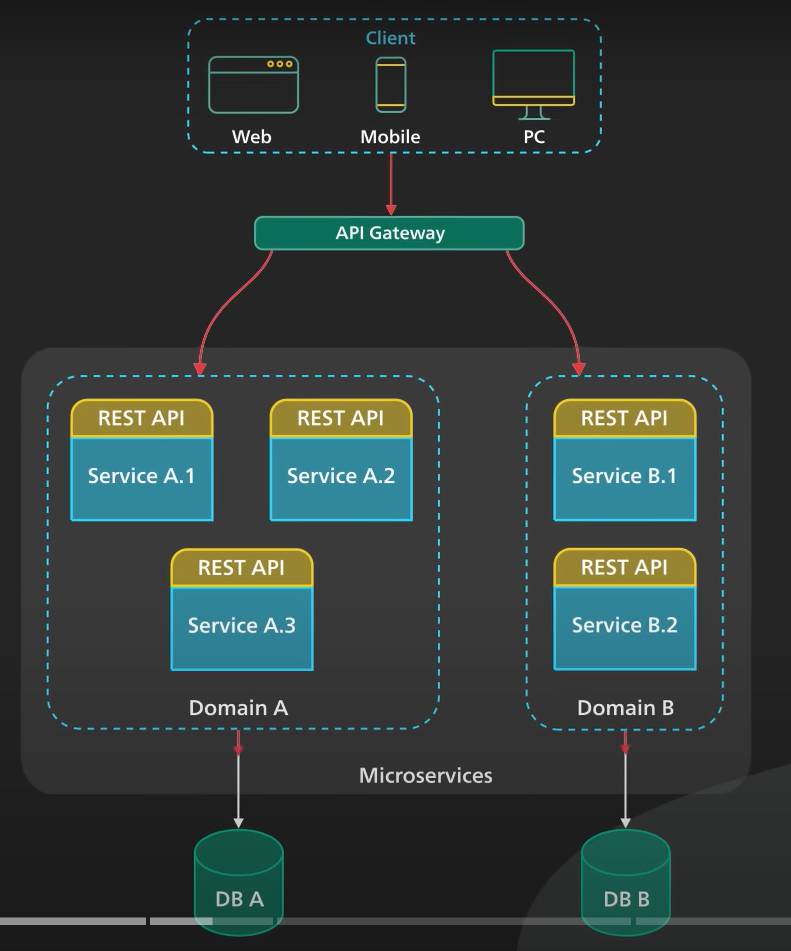
Hackers are always looking for ways to undermine the security of the web. To this day, **cross-site scripting (XSS)** is one of the most popular ways of bypassing the same origin policy and accounted for 84% of all security vulnerabilities not long ago.



**---------------------------------------------------------------------------------------------------------------------**

**API GATEWAY:**

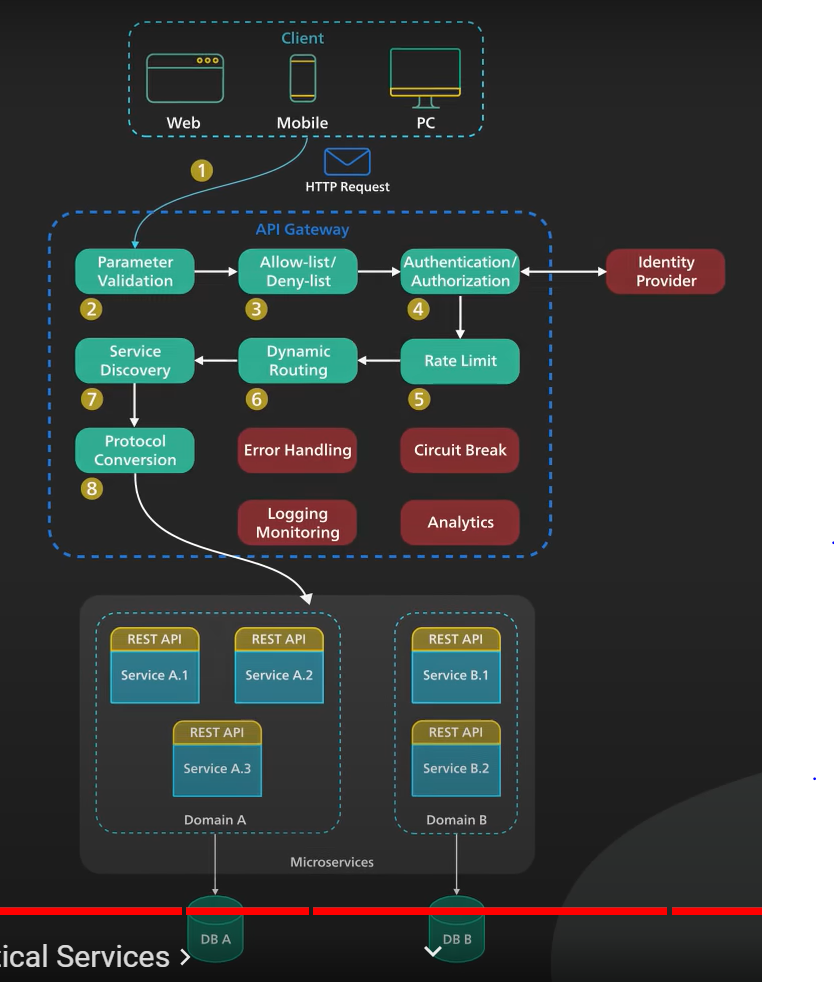
An API gateway is a single point of entry to the clients of an application. It sits between the clients and a collection of backend services for the application.



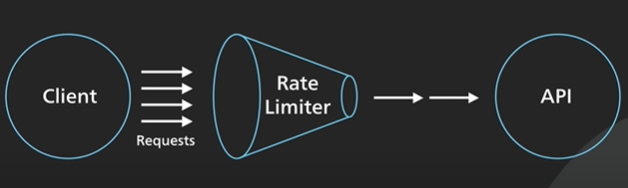
Functions provided by API gateway:

1. Authentication and security policy enforcement
2. Load balancing and circuit breaking
3. Protocol translation and service discovery
4. Monitoring, logging, analytics, and billing
5. Caching

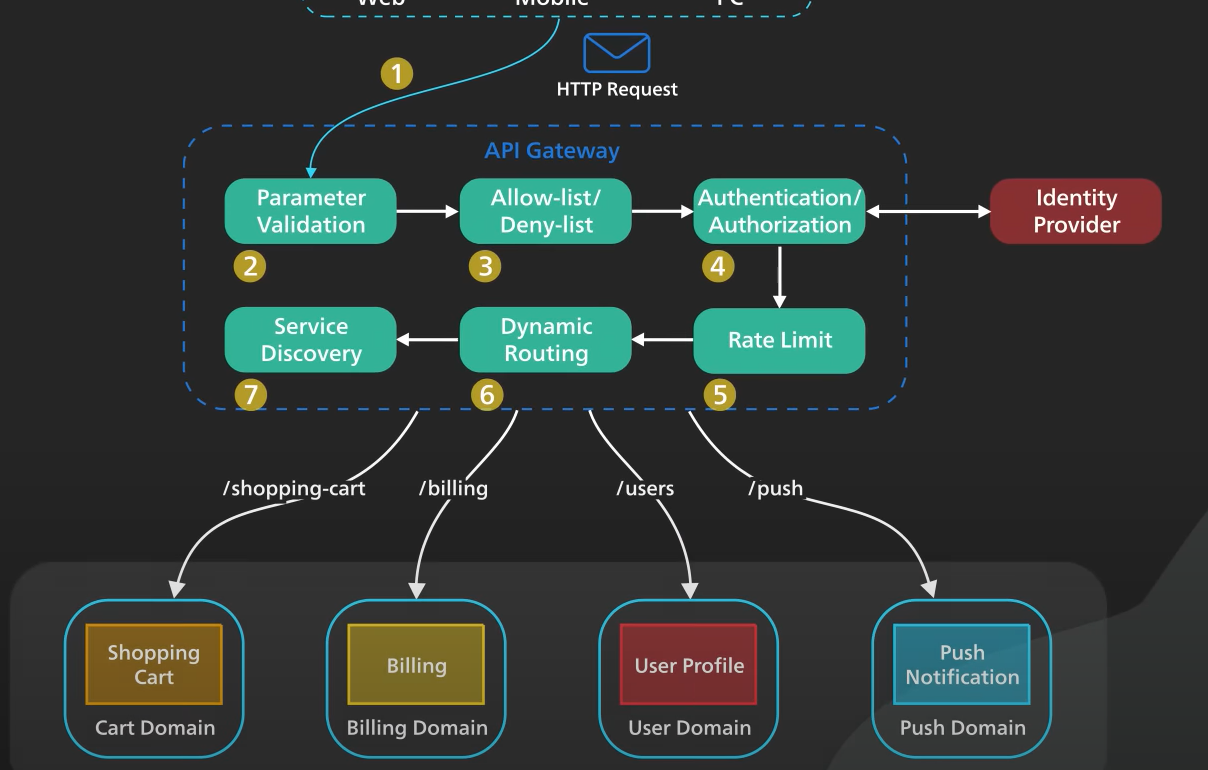
Flow of API Gateway:



1. The client sends a request to the API gateway. The request is typically HTTP-based. It could be REST, GraphQL, or some other higher-level abstractions.
2. the API gateway validates the HTTP request.
3. the API gateway checks the caller’s IP address and other HTTP headers against its allow-list and deny-list. It could also perform basic  rate limit checks against attributes such as IP address and HTTP headers. For example, it could reject requests from an IP address exceeding a certain rate.



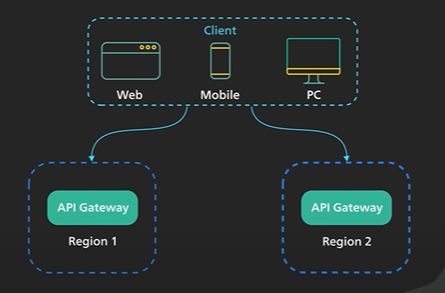
1. the API gateway passes the request to an identity provider for authentication and authorization. The API gateway receives an authenticated session back from the provider with the scope of what the request is allowed to do.
2. a higher-level rate-limit check is applied against the authenticated session. If it is over the limit, the request is rejected at this point.
3. Dynamic routing helps route client requests to different microservices based on specific parameters, such as the request URL, HTTP header, or request payload.
4. With the help of a service discovery component, the API gateway locates the appropriate backend service to handle the request by path matching.



1. the API gateway transforms the request into the appropriate protocol and sends the transformed request to the backend service. An example would be gRPC. When the response comes back from the backend service, the API gateway transforms the response back to the public-facing protocol and returns the response to the client.

A proper API gateway also provides other critical services. For example, an API gateway should track errors and provide circuit-breaking functionality to protect the services from overloading. An API gateway should also provide logging, monitoring, and analytics services for operational observability.

An API gateway is a critical piece of the infrastructure. It should be deployed to multiple regions to improve availability. For many cloud provider offerings, the API gateway is deployed across the world close to the clients.



---------------------------------------------------------------------------------------------------------------------

**Azures APIM**

**JWT Token :**

Part 1. <https://www.youtube.com/watch?v=p_sDlCyzUFU>

Part 2. <https://www.youtube.com/watch?v=bIAbbhLwosg>

**REFERENCES:**

1. API Gateway: <https://www.youtube.com/watch?v=6ULyxuHKxg8>