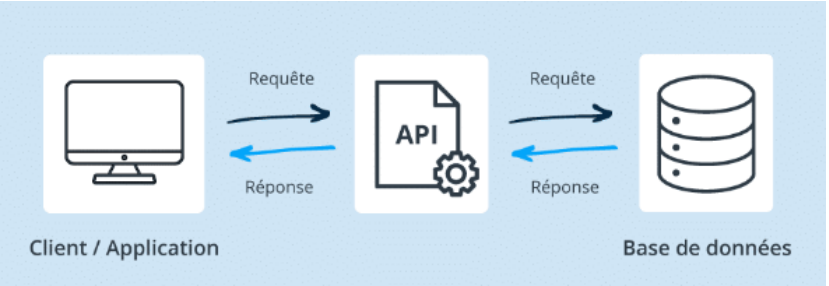
API Documentation

Understanding APIs

Is a way for two systems , usually a client and server to communicate over the Internet with **HTTP requests.**

It consists of endpoints (specific URLs) where services or resources can be accessed, and it typically uses various HTTP methods to perform operations. APIs facilitate the exchange of messages, which are structured data packets that include a request sent from the client to the server and a response returned from the server to the client. These messages can be formatted in different ways, such as JSON, XML, or Protobuf, depending on the API design. A key characteristic of APIs is that they can be either stateless or stateful, affecting how they manage session information and interactions over time.

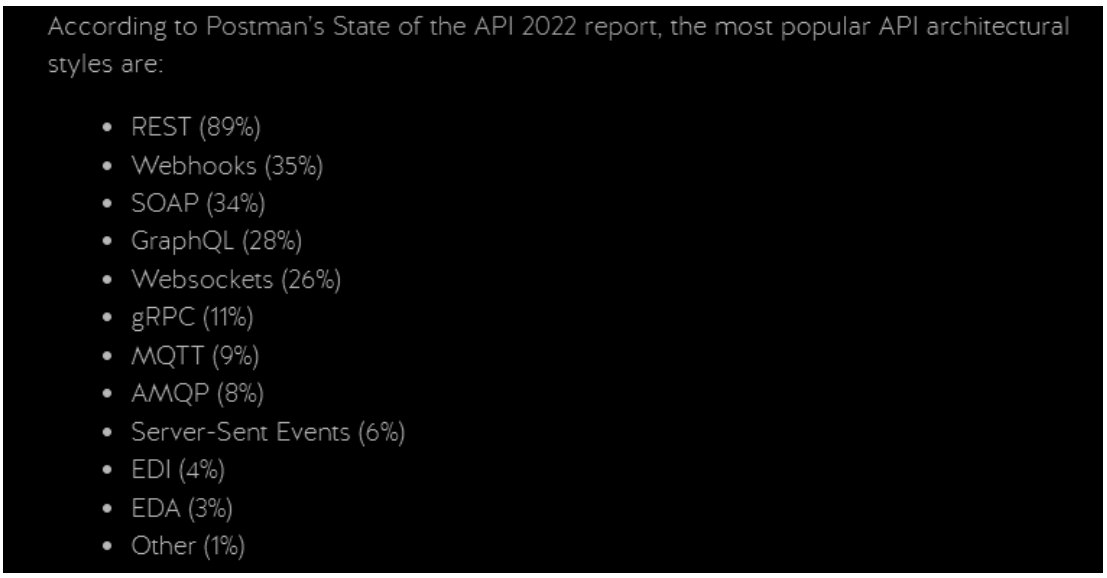
API : Application Programming interface
Check out this link for more details : Resource (1)



Common API Types

There are a lots of API's type ,the most common APIs include **REST APIs** for web services, widely used by platforms like Twitter and GitHub; **GraphQL APIs** for efficient data querying, popular with Facebook and Shopify; and **SOAP APIs**, often used in enterprise environments for secure transactions like banking. REST is the most widely adopted due to its simplicity.

Here an example of the most popular other API architectural styles :



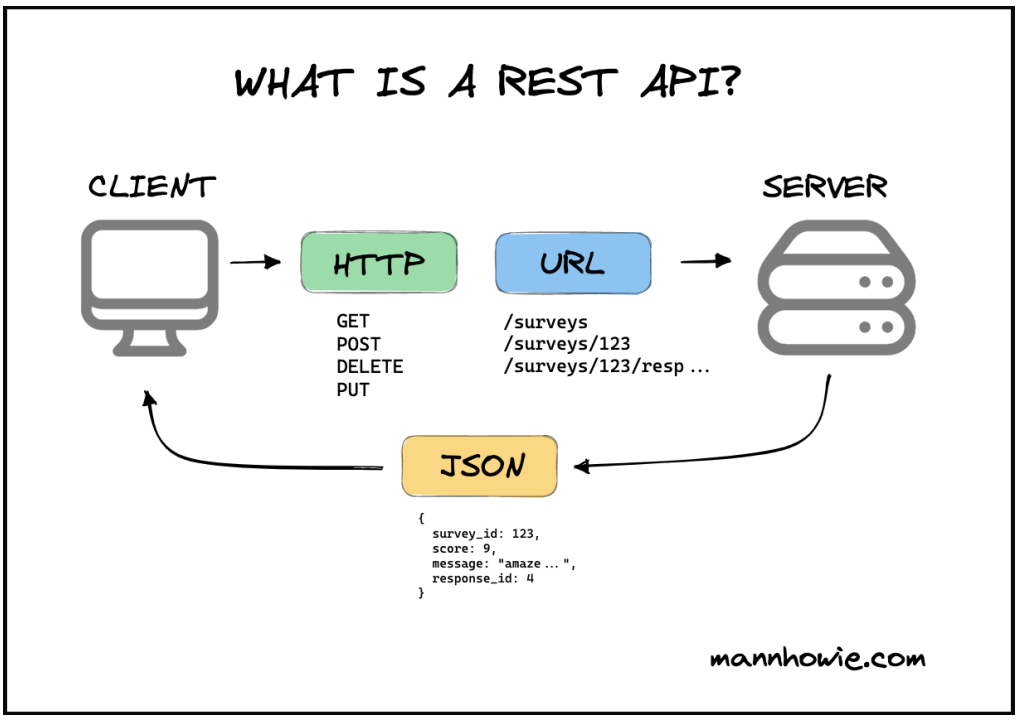
Introduction to REST APIs

What REST API for ?

A **REST API** allows systems to communicate over the web by treating data as resources, each identified by a unique URL. It uses **HTTP** methods like :

- **GET** to retrieve data
- **POST** to create
- **PUT** to update
- **DELETE** to remove resources.

REST APIs are stateless, meaning each request contains all the necessary information. Data is often exchanged in formats like **JSON** or **XML**.



- **Identificateur de ressources uniformes (URI)** : parfois appelé Uniform Resource Locator (URL), identifie la ressource que le client souhaite manipuler.

Exemple de requête :

Différents composants d'un URI (Uniform Resource Identifier)

REST is a set of architectural constraints, not a protocol or a standard. API developers can implement REST in a variety of ways.

When a client request is made via a RESTful API, it transfers a representation of the state of the resource to the requester or endpoint. This information, or representation, is delivered in one of several formats via **HTTP**: **JSON** (Javascript Object Notation), **HTML**, **XLT**, **Python**, **PHP**, or **plain text**. JSON is the most generally popular file format to use because, despite its name, it’s language-agnostic, as well as readable by both humans and machines.

Something else to keep in mind: Headers and parameters are also important in the HTTP methods of a RESTful API HTTP request, as they contain important identifier information as to the request's metadata, authorization, uniform resource identifier (URI), caching,

cookies, and more. There are request headers and response headers, each with their own HTTP connection information and status codes.

In order for an API to be considered RESTful, it has to conform to these criteria:

- A client-server architecture made up of clients, servers, and resources, with requests managed through HTTP.
- **Stateless** client-server communication, meaning no client information is stored between get requests and each request is separate and unconnected.
- Cacheable data that streamlines client-server interactions.
- A uniform interface between components so that information is transferred in a standard form. This requires that:
    - resources requested are identifiable and separate from the representations sent to the client.
    - resources can be manipulated by the client via the representation they receive because the representation contains enough information to do so.
    - self-descriptive messages returned to the client have enough information to describe how the client should process it.
    - hypertext/hypermedia is available, meaning that after accessing a resource the client should be able to use hyperlinks to find all other currently available actions they can take.
- A layered system that organizes each type of server (those responsible for security, load-balancing, etc.) involved the retrieval of requested information into hierarchies, invisible to the client.
- Code-on-demand (optional): the ability to send executable code from the server to the client when requested, extending client functionality.

Though the REST API has these criteria to conform to, it is still considered easier to use than a prescribed protocol like SOAP (Simple Object Access Protocol), which has specific requirements like XML messaging, and built-in security and transaction compliance that make it slower and heavier.

In contrast, REST is a set of guidelines that can be implemented as needed, making REST APIs faster and more lightweight, with increased

scalability—perfect for Internet of Things (IoT) and mobile app development.
Article (3)

API synchrones et asynchrone

API Asynchrones

If the API request takes some processing time for the server or if the data is not readily available, it is best to use an asynchronous API.

The client application should then be able to be notified or inquire when the result of its API request is available.

Advantages: The client application can continue execution without being blocked for the time it takes for the server to process the request. The client application can have better performance because it can multitask and make other queries.

Disadvantage: Unnecessary or excessive use of asynchronous calls can have the opposite effect on performance.

API synchrones

API are typically designed to be synchronous when the data in the request is readily available, such as when the data is stored in a database or in internal memory. The server can immediately retrieve this data and respond immediately.

The client application that makes the API request must wait for the response before performing additional code execution tasks.

Advantages: The app receives the data immediately.

Disadvantage: If the API is not properly designed, it will be a bottleneck because the application has to wait for the response.

The differences between a URI and a URL

URI	URL
URI est l'acronyme d'Uniform Resource Identifier.	URL est l'acronyme de Uniform Resource Locator.
L'URI est le sur-ensemble de l'URN et de l'URL.	L'URL est le sous-ensemble de l'URI.
L'URI identifie une ressource et la différencie des autres en utilisant un nom, un emplacement ou les deux.	L'URL identifie l'adresse web ou l'emplacement d'une ressource unique.
L'URI contient des composants tels qu'un schéma, une autorité, un chemin et une requête.	L'URL a des composants similaires à l'URI, mais son autorité est constituée d'un nom de domaine et d'un port.
Un exemple d'URI est **ISBN 0-476-35557-4.**	Un exemple d'URL serait **https://hostinger.com.**
L'URI est généralement utilisé dans les fichiers XML, les fichiers de bibliothèque de balises et d'autres fichiers, tels que JSTL et XSTL.	L'URL sert principalement à rechercher des pages web sur Internet.
Le schéma de l'URI peut être un protocole, une spécification ou une désignation telle que HTTP, fichier ou données.	Le schéma de l'URL est un protocole, tel que **HTTP et HTTPS**.

URL syntax

Each URL must follow URI syntax, which is similar in structure to a URI.

Below is an example of URL syntax::`https://www.example.com/forum/questions/?tag=networking&order=newest#top`

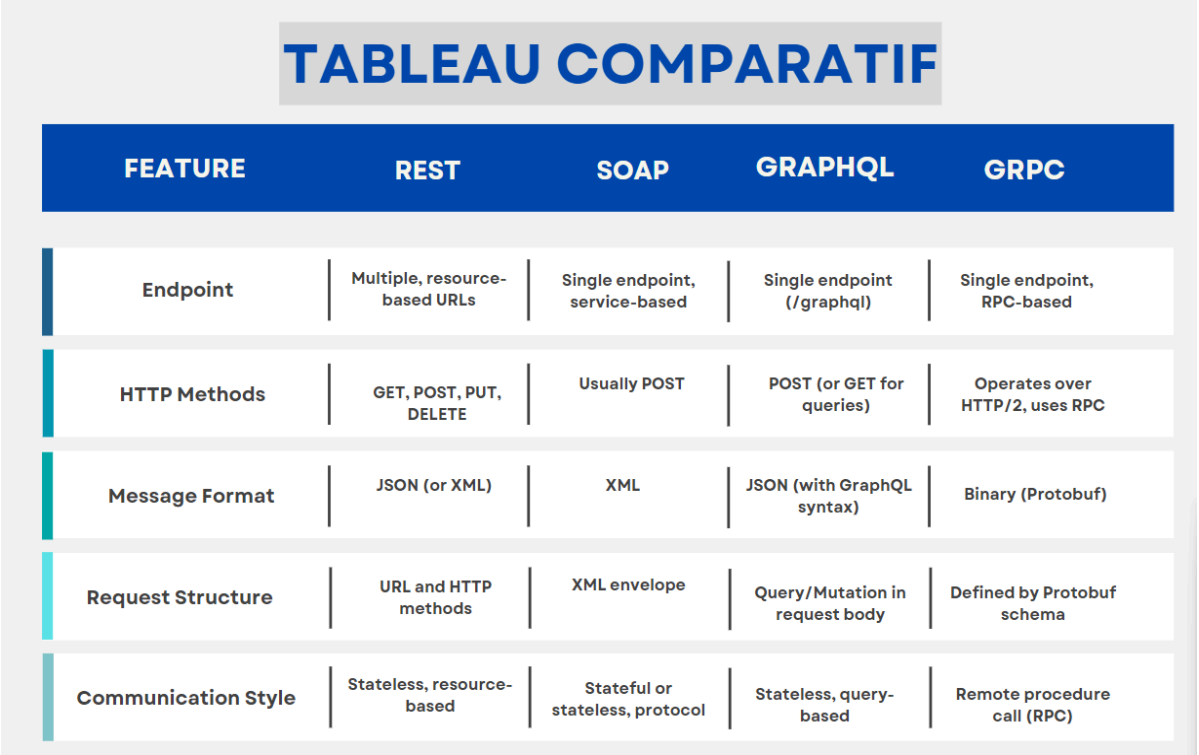
URI syntax

The syntax of a Uniform Resource Identifier (URI) defines its structure, which allows a program to understand it.

Here is the generic syntax of a URI:

```
scheme://authority]path[?query][#fragment]
```

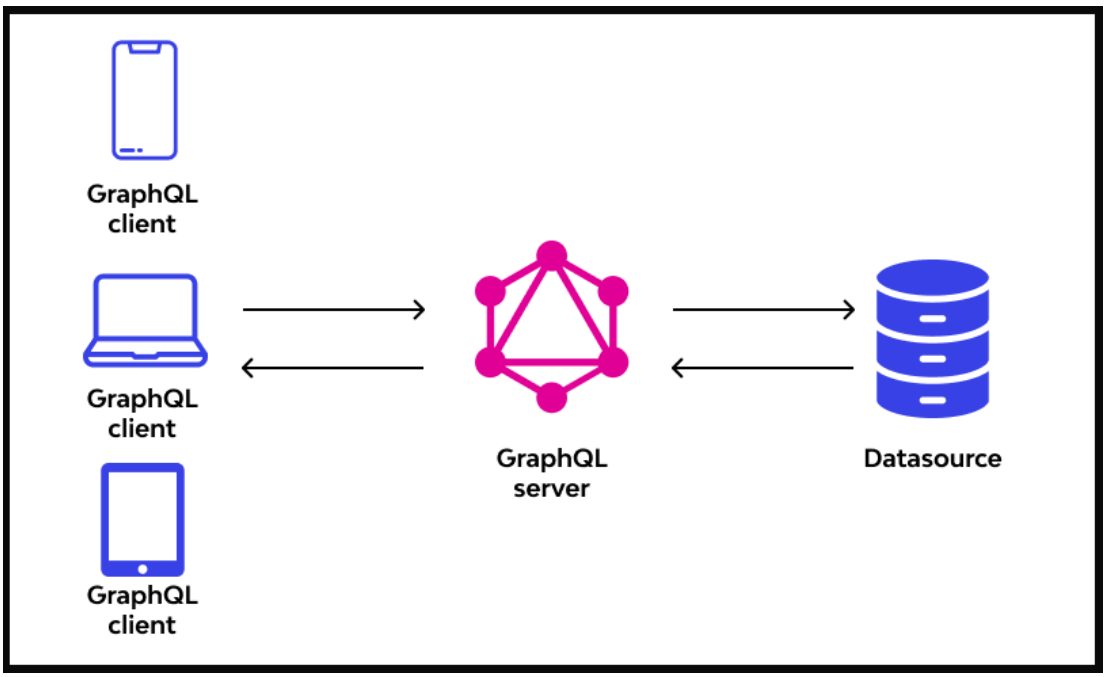
Key Differences Between REST, SOAP, GraphQL, and gRPC:



FEATURE	REST	SOAP	GRAPHQL	GRPC
Endpoint	Multiple, resource-based URLs	Single endpoint, service-based	Single endpoint (/graphql)	Single endpoint, RPC-based
HTTP Methods	GET, POST, PUT, DELETE	Usually POST	POST (or GET for queries)	Operates over HTTP/2, uses RPC
Message Format	JSON (or XML)	XML	JSON (with GraphQL syntax)	Binary (Protobuf)
Request Structure	URL and HTTP methods	XML envelope	Query/Mutation in request body	Defined by Protobuf schema
Communication Style	Stateless, resource-based	Stateful or stateless, protocol	Stateless, query-based	Remote procedure call (RPC)

Overview of GraphQL APIs

A **GraphQL API** is a query language for APIs that allows clients to request exactly the data they need, making it more efficient and flexible than traditional **REST APIs**. With **GraphQL**, clients can fetch multiple resources in a single request and specify which fields they want. It uses a strong type system defined by a schema, making it easier to validate and document data. Unlike **REST**, where endpoints return fixed data structures, **GraphQL** allows clients to shape the response to their needs.



A GraphQL service is created by defining **types** and **fields** on those types, then providing functions for each field on each type. For example, a **GraphQL** service that tells you who the logged in user is (me) as well as that user's name might look like this:

```
type Query {
  me: User
}

type User {
  id: ID
  name: String
}
```

Along with functions for each field on each type:

```
function Query_me(request) {
  return request.auth.user
}

function User_name(user) {
  return user.getName()
}
```

For example, the query:

```
{
  me {
    name
  }
}
```

Could produce the following JSON result:

```
{
  "me": {
    "name": "Luke Skywalker"
  }
}
```

SOAP APIs Explained

A **SOAP API (Simple Object Access Protocol)** is a protocol used for exchanging structured information in web services. It relies on **XML** for message formatting and typically runs over **HTTP** or **SMTP**. **SOAP** is more rigid and secure compared to **REST**, often used in enterprise-level applications like banking or payment systems. It has built-in standards for security (WS-Security) and transactions, making it suitable for scenarios where reliability and formal contracts are essential.

In order to call a **SOAP API**, you'll most likely need to include a **SOAP** library with your programming language. Although it's possible to make **SOAP API** calls without **SOAP** libraries, it's more efficient to work with an abstraction rather than crafting the messages yourself. The **SOAP** messages are verbose, mainly due to reliance on **XML**.

To retrieve a user profile from a fictitious **SOAP API**, you might make the following request using the **Zeep** library:

```
from zeep import Client

client = Client('https://www.example.com/exampleapi')
result = client.service.GetUser(123) # request user with ID 123

name = result['Username']
```

Let's look at how this **SOAP** call might be structured:

```xml
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="https://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetUser>
      <m:UserId>123</m:UserId>
    </m:GetUser>
  </soap:Body>
</soap:Envelope>
```

And the response might look something like this:

```xml
<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="https://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="https://www.w3.org/2003/05/soap-encoding">

<soap:Body>
  <m:GetUserResponse>
    <m:Username>Tony Stark</m:Username>
  </m:GetUserResponse>
</soap:Body>

</soap:Envelope>
```

Postman: API Testing Tool

Introduction

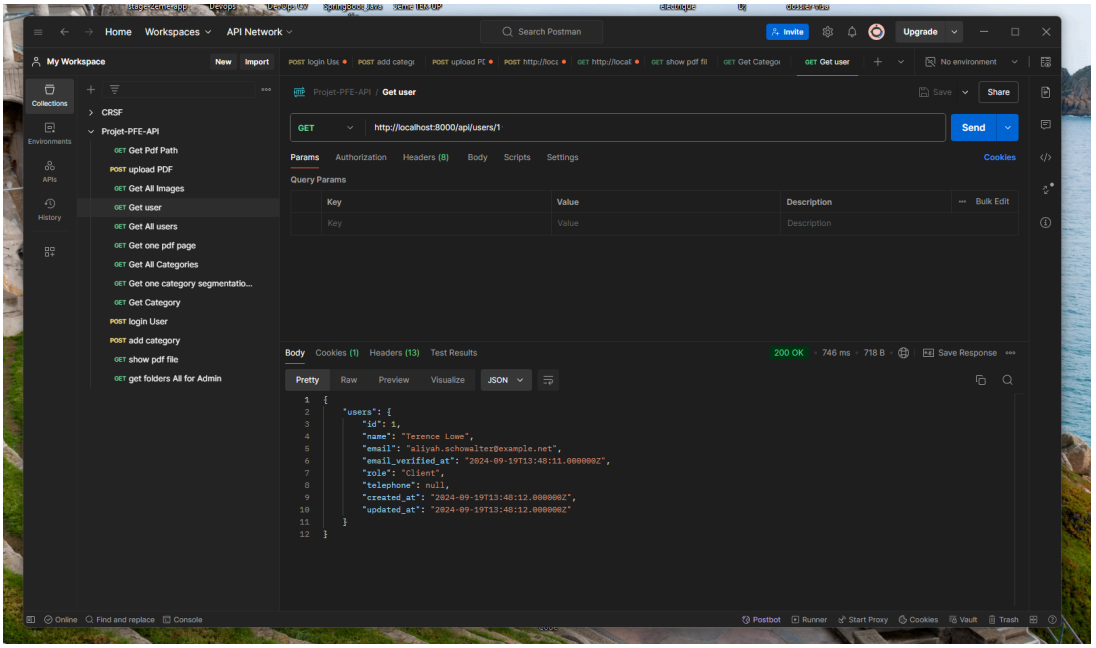
Postman is a popular API development tool that allows developers to design, **test**, and **document** APIs. It provides a user-friendly interface to make HTTP requests, which can be useful for interacting with APIs during development.

API Testing

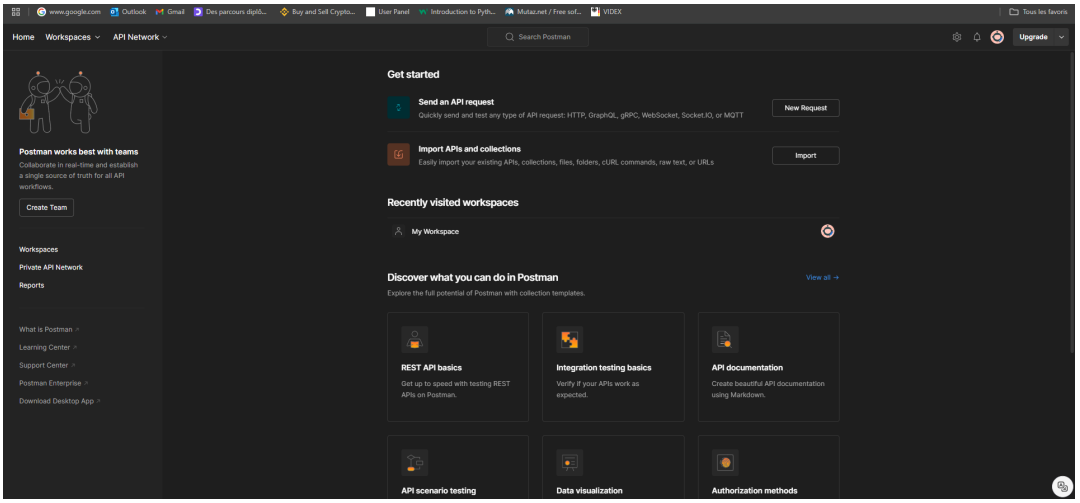
Users can create and run automated tests on APIs to ensure they function as expected.

Postman offers both a web version and a local agent, enabling seamless API development and testing directly from your browser or as a standalone application on your machine.

that’s the postman local agent



and that is the postman platform

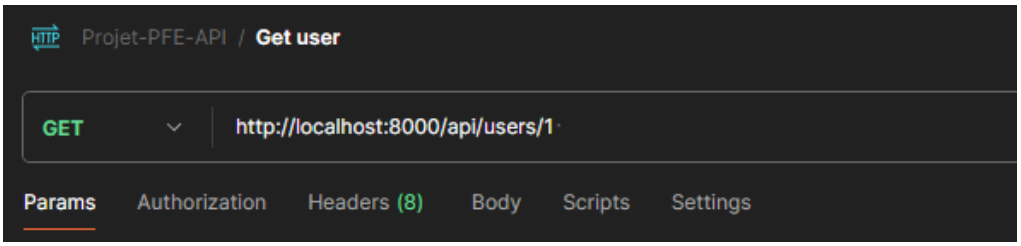


those are some features of **Postman**

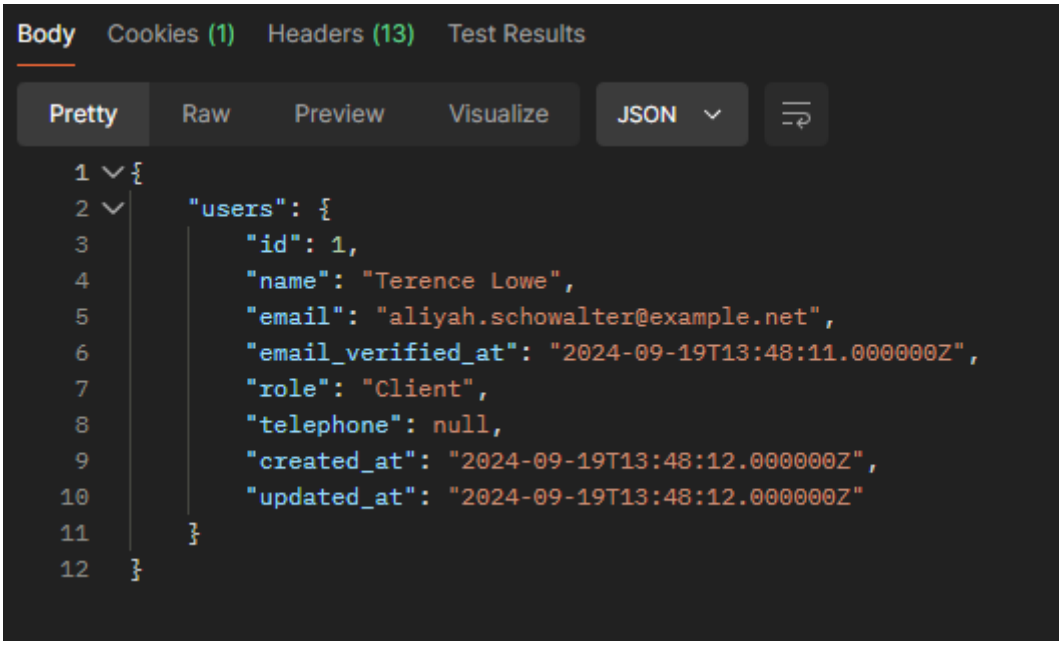
- API Testing Made Easy
- Build and Send HTTP Requests
- Automate Your API Tests
- Organize Requests with Collections
- Manage Multiple Environments
- Share and Collaborate on API Projects

- Simulate APIs with Mock Servers
- Monitor API Performance
- Generate API Documentation
- Debug and Inspect API Responses

We trying in this example to get **REST API** request from server



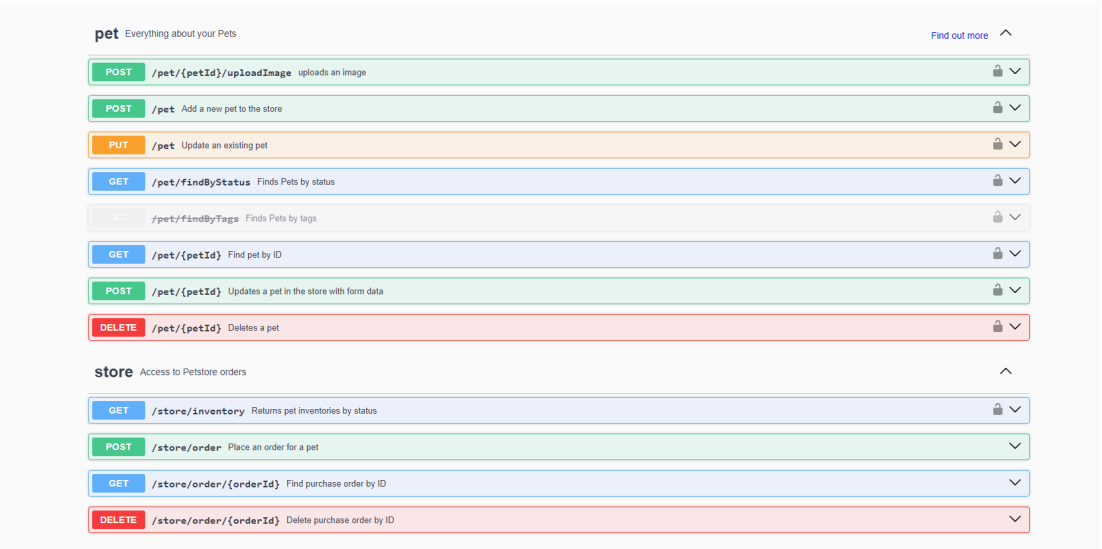
The response should look like this



Swagger: API Documentation and Testing

Swagger is a framework for **designing**, **building**, **documenting**, and **consuming RESTful APIs**. It provides a set of tools and a specification format that makes it easier to define **APIs** in a structured way, allowing both developers and consumers to understand and interact with the **API** effectively.

here is an example describe how swagger manage your API's and present them in good looking style



API Authentication and Security

What is API authentication?

API authentication is the process of verifying the identity of a user who is making an **API request**, and it is a crucial pillar of API security. There are many types of **API authentication**, such as **HTTP** basic authentication, **API key authentication**, **JWT**, and **OAuth**, and each one has its own benefits, trade-offs, and ideal use cases. Nevertheless, all **API authentication** mechanisms share the goal of protecting sensitive data and ensuring the **API** is not misused.

What are the benefits of API authentication?

Today, an increasing number of organizations are focusing on APIs in order to unlock new features and advance business objectives. In fact, many teams have adopted the API-first development model, in which applications are conceptualized and built as a collection of services that are delivered through APIs. With this approach, teams prioritize **API quality** and **security** in order to ensure that their APIs remain **highly performant** and scalable—without serving as entry points for attackers. **API authentication** is a primary way in which **APIs** are secured, and it enables teams to protect sensitive data, build trust with users, and safeguard the company's reputation.

Best 7 best practices of successful API teams for API Owner’s

Best practice #1: Focus relentlessly on the value of the API (Focus on Value)

API programs must prioritize their core goal of delivering value while avoiding complexity. The value proposition determines user utility, a crucial API success driver. A compelling value proposition is essential for effective marketing. Aligning it with corporate objectives ensures sustainability, allowing established companies to enhance their offerings through APIs.

in short terms : Always prioritize what the API brings to users. Understand their needs and how the API can help.

Best practice #2: Make the business model clear from the beginning (Clear Business Model)

Creating a successful API goes beyond a value proposition alone. It requires pairing the API with a well-defined business model. While recognizing and conveying the API’s value is important, its costs must also be balanced against its financial or tangible benefits. In his crowd-created book Business Model Generation, Alex Osterwalder defines the business model of an organization as the way the organization proposes, creates, delivers, and captures value.

in short terms : Define how the API will generate value and cover costs from the start.

Best practice #3: Design and implement with the user in mind (User-Centric Design)

Simplicity in API design is context-dependent, with designs varying in complexity across use cases. Striking the right balance in API method granularity is essential. Simplicity can be considered on different levels:

in short terms : Design APIs with the user in mind, ensuring they are simple and flexible to use.

Best practice #4: Place API operations at the top of the list (Prioritize Operations)

The API platform team manages APIs once they are live to make sure that they are accessible and delivered according to developers' expectations. While vendors offer solutions, selecting the right strategy is essential for success. API management involves 2 main functions:

1. Streamlining internal processes to be efficient and reduce cost.
2. Making operations effective to meet the expectations of external developers to the program

API operations donut

1. Dependability. Ensure API availability through redundancy or quotas and rate limits. These align with business models and prevent downtime.
2. Flexibility. Offer technical and business adoption options, such as changing between price plans or cancellations. Keep in mind greater flexibility can translate into higher internal efforts and costs.
3. Quality. Maintain consistent adherence to developer expectations using service level agreements (SLAs) and streamlined processes.
4. Speed. Attain low latency and high throughput with techniques like throttling and caching, potentially aligned with business models.
5. Cost. Optimize developer value while minimizing internal costs without compromising quality

in short terms : Manage API operations effectively to ensure reliability and meet user expectations.

Best practice #5: Obsess about developer experience (Enhance Developer Experience)

While developer experience may sound like it is about API design, it goes far beyond that. Think of developer experience as the packaging and delivery of the API rather than the API itself. You can have a wonderfully designed REST or containerized API, but if it is hard to sign up for access, read documents, and test, then you have created a poor developer experience that can significantly affect your API's success

One of the pioneers in the field of APIs and API management, John Musser described several ways to enhance engagement that still hold true:

1. Clearly define the API's purpose.
2. Offer easy sign-up and free access.
3. Communicate pricing transparently.
4. Provide thorough documentation.

in short terms : Make it easy for developers to use the API with clear documentation and support.

Best practice #6: Go beyond marketing 101 (Effective Marketing)

Marketing APIs to developers can be challenging, especially if the value presented does not match the developer's business or technical needs. APIs should be marketed like any other product, aligning with segmentation, targeting, and positioning (STP). Effective marketing matches the right API with the right developers, moved by API value and STP principles.

1. Segmentation. Categorize customers as internal users, partners, end users, or external developers. For effective engagement, segment the vast developer market using the jobs-to-be-done method.

2. Targeting. Assess segment attractiveness based on accessibility, substantiality, and differentiation. Select promising segments and tailor marketing tactics accordingly.

3. Positioning. Make your API stand out by addressing specific needs, relieving pain points, and providing gains for chosen developer segments

in short terms : Market the API like a product, targeting the right audience and clearly communicating its benefits.

Best practice #7: Remember API retirement and change management (Plan for Changes)

API advice tends to focus heavily on API design, creation, and operation. However, one of the most critical segments of the API journey that gets overlooked is what happens many months after launch and operation—managing updates to the API, including API deprecation. Disruptions from abrupt changes can erode confidence and entail significant costs, especially with unknown developers, mobile application approvals, or devices lacking update capabilities. API changes are typically classified as nonbreaking or breaking. Nonbreaking changes include new methods and enhancements, while breaking changes involve removal, modifications, or total depreciation. Major version numbers and migration plans address breaking changes; minor versions handle nonbreaking ones. It can be challenging to make sure that changes won't break some applications, so we strongly recommend that any changes to the API, even if categorized as nonbreaking, should be rolled out by:

- Providing a test endpoint with the new version before launch.
- Sending an email or other communication to developers informing them of the change and giving timing and details.

Effective communication and transparent contracts are vital. Share problem details, uphold commitments, and outline version support duration. For example, deprecating an API requires a structured approach, including extended notice, addressing media coverage, migration plans, and data export tools if needed.

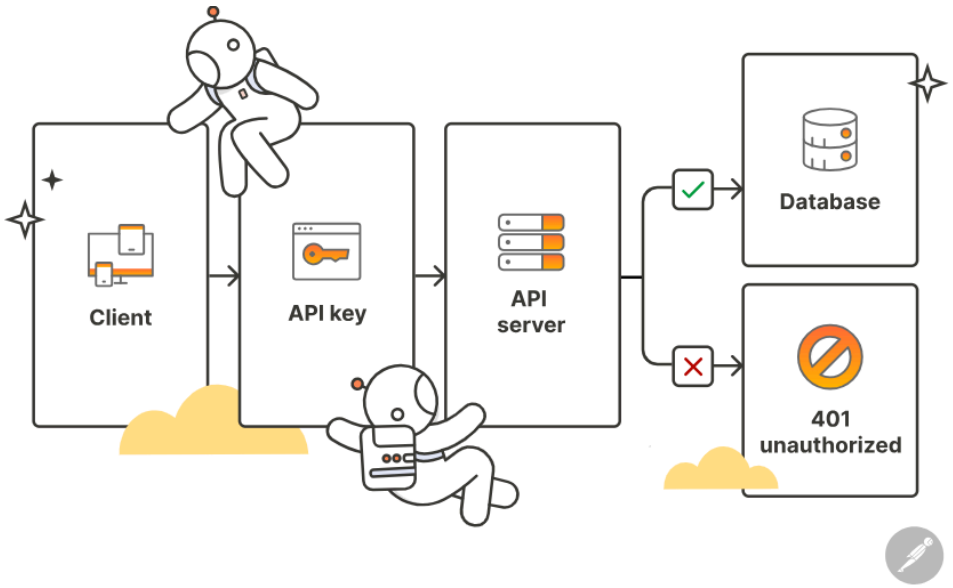
1. Introduce a new version for testing.
2. Notify users about retiring the old version.
3. Assist users during the transition.
4. Retire the old version.

in short terms : Have a strategy for updating or retiring APIs to maintain trust and minimize disruptions.

Visit Ressource (4) for more details.

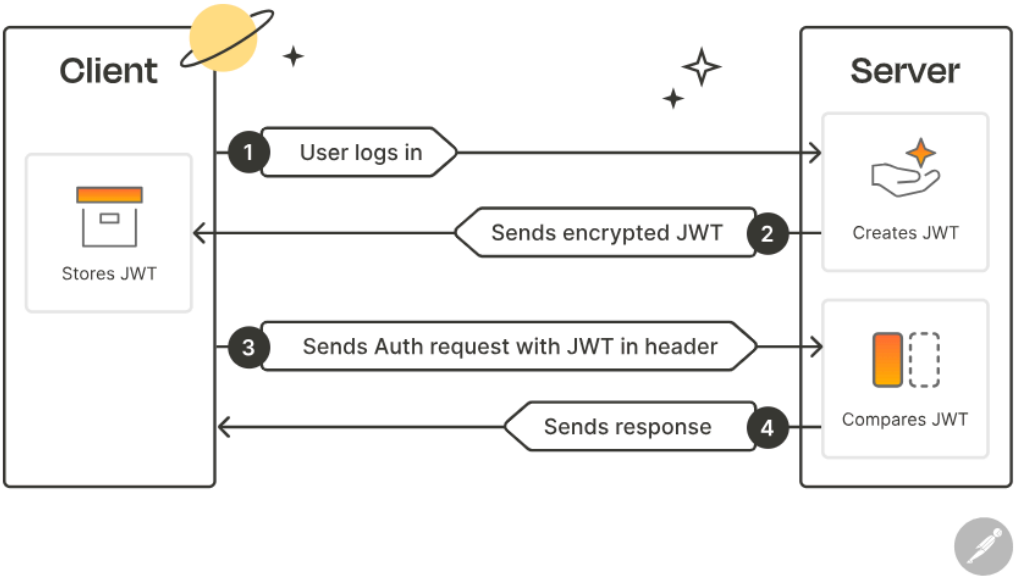
API authentication, and how do they work?

API key authentication



An API key is a unique identifier that an API provider issues to registered users in order to control usage and monitor access. The API key must be sent with every request—either in the query string, as a request header, or as a cookie. Like HTTP basic authentication, API key authentication must be used with HTTPS to ensure the API key remains secure.

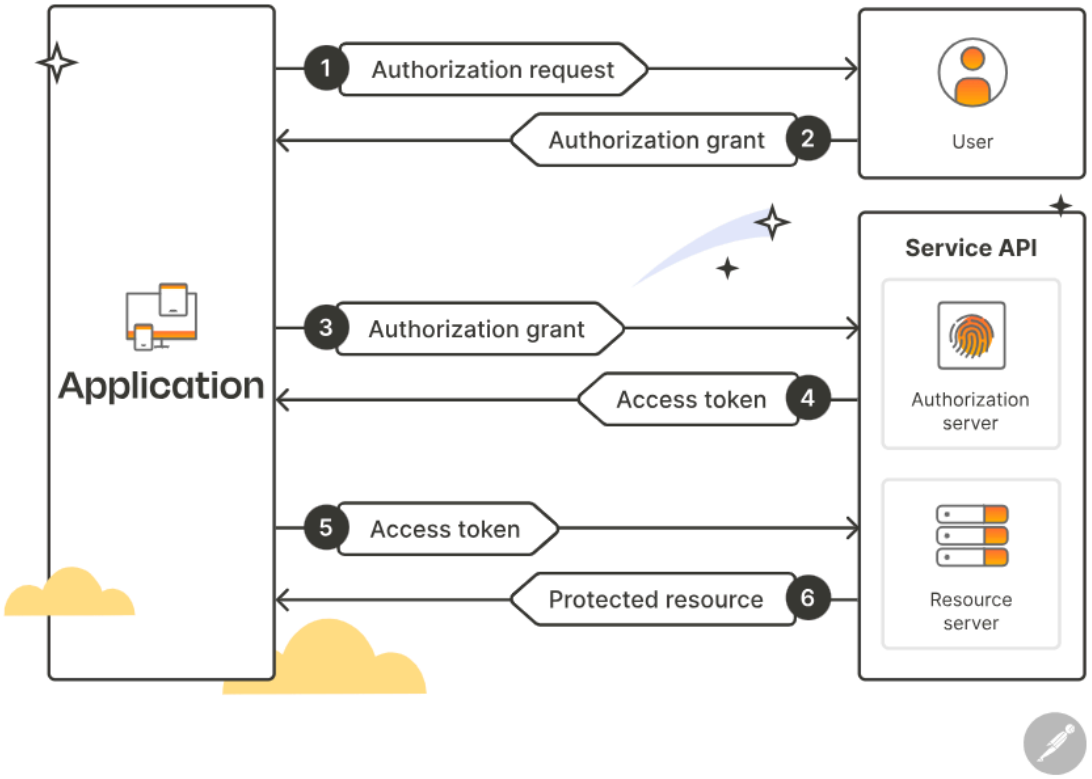
JWT authentication



JWT, which stands for JSON Web Token, is a compact, stateless mechanism for API authentication. When a user logs into an application, the API server creates a digitally signed and encrypted JWT that includes the user's identity. The client then includes the JWT in every subsequent request, which the

server deserializes and validates. The user's data is therefore not stored on the server's side, which improves scalability.

OAuth authentication



OAuth is a token-based authentication mechanism that enables a user to grant third-party access to their account without having to share their login credentials. OAuth 2.0, which provides greater flexibility and scalability than OAuth 1.0, has become the gold standard for API authentication, and it supports extensive API integration without putting user data at risk.

API Security Checklist

Introduction

APIs come in many flavors, including REST, SOAP, graphQL, gRPC, and WebSockets, and each has its own use cases and common vulnerabilities. The issues covered in this checklist can occur in any kind of API. Regardless of which

technology you have used to implement your API, read on to find out what you can do today to address the biggest potential risks associated with it.

Improper API asset management and discovery

APIs present a large attack surface as each action requires its own endpoint. Ensuring all endpoints are identified and fully documented allows for potential attack vectors to be recognized and mitigated or monitored.

Keep track of your assets and infrastructure to ensure they are properly secured

- **Make an inventory of all API infrastructure** (from testing to production), including who can access each infrastructure item and what data it contains, and which API functions access them or are hosted by them
- Continuously run **API Discovery** to identify changes in APIs and surface shadow APIs and rogue APIs
- Do the same for all services that are **integrated** with your API
- Thoroughly document **every aspect of your API**, including all authorization policies, error reporting, and security measures. Share this documentation with the team members responsible for testing and reviewing the security of the application
- To prevent unintentional data disclosure, avoid using production data for testing unless doing so is **absolutely necessary**

API abuse, lack of resources and rate limiting

APIs that do not impose rate or resource limits are vulnerable to brute force or other DoS style attacks. For example, brute force attacks are common against authentication endpoints and make it easy for attackers to perform password stuffing or user enumeration attacks. In addition, attackers can use DoS techniques to overload API infrastructure.

Limiting access to your resources

- **Implement rate limits** for every API endpoint that limit how many requests a client can make in a given period of time;
- **Implement** request size limits and limits to the size of submitted strings and arrays;

- Validate user submitted data **before** it is executed by your API functions;
- Use bot mitigation tools to **prevent abuse** by automated tooling;
- **Block traffic** from unwanted geographical regions, data centers, and Tor relay nodes;
- Only allow traffic to private API endpoints from **allow-listed** IP addresses;
- Keep known continuous attackers **in the block list.**

Injections

Unsanitized input from a malicious client can be used to execute arbitrary code on your infrastructure. SQL injections can result in attackers having full access to production databases, and shell injections have the potential to grant attackers control over application servers.

- Validate and sanitize **all input** from the client;
- Ensure all input is **properly escaped** using the correct syntax;
- Use **API threat prevention tooling** that supports required protocols (REST, GraphQL, etc). Ensure that your solution provides poproper inspection for API calls and is a good fit for detecting injections;
- Utilize your OpenAPI/Swagger schema **to validate incoming API requests** and block malicious requests (for example, by using an open source validator)

Broken object level authorization (BOLA) / Insecure Direct Object Reference (IDOR)

Object Level Access Control issues arise if authorization checks are not performed for every function that could potentially be manipulated via user input by an untrusted party. For example, if an object is accessed by a unique ID specified by the end user, an attacker could change this ID in a malicious request to gain access to other objects the user should not have access to.

Steps you can take to fix broken authorization

- Implement an **authorization mechanism** that checks whether the logged in user has permission to perform an action

- Use this authorization mechanism **in all functions** that accesses sensitive data

- Use **randomly generated GUIDs** (as they are hard to guess) as object identifiers for user requests.

Broken user authentication

Improperly implemented user authentication often renders other security measures obsolete as it does not provide a foundation of an authenticated user to build other security measures with. Technical flaws in a user authentication system can allow malicious parties to impersonate legitimate users. Some technical flaws could include using expired or leaked tokens/sessions, guessable or predictable authentication tokens, or otherwise broken credential verification before minting valid user sessions.

Protecting your API against attacks on your authentication system

- Use commonly accepted standards **like OAuth and JWT** for the authentication process

- **Identify and document** all paths that can be used to authenticate with your API and ensure they are reviewed for possible credential leaks

- Do not return any **sensitive** information like passwords, keys, or tokens directly in API responses

- **Protect** all login, password recovery, and registration paths using rate limiting, brute force protection, and by adding lockout measures for abusive traffic sources

- Implement and use **multi-factor authentication** (MFA) wherever possible, and use revocable tokens where implementing MFA is not feasible.

Excessive data exposure

For the sake of convenience, many developers expose all of the properties of objects through API endpoints. This is intended to allow front-end developers access to all of the required resources, but can result in unintended data exposure.

What your engineers can do to avoid unintended data exposure

- Define exactly which object properties are to be returned in your **API functions** rather than returning entire objects
- Return **only the data the client requests** from your API functions rather than returning all available data and expecting the client to filter it
- **Limit the number of records** that can be affected by a query in API functions to prevent mass updating or disclosure of database records

- **Validate** API responses from a central schema that filters out object properties that should not be visible to the requesting user.

Broken function level authorization

Overly complex and decentralized authorization policies make it confusing for engineers to implement the correct authorization for a given object or endpoint, leading to mistakes in authorization that can be exploited to access protected resources.

Important steps to fix authorization issues in your code

- Ensure your authorization frameworks grant access **explicitly** to individual resources.
- Ensure the default permission for all users for all resources is to **deny access**.
- **Centralize** your authorization code so that it can be regularly reviewed and vetted, knowing that the review covers authorization wherever it is used in your API.

Mass assignment

Mass assignment flaws allow attackers to modify objects by guessing property names or endpoint addresses that shouldn't be accessible, or by providing additional object properties through object relationships.

Prevent mass assignment attacks by implementing measures to validate input

- **Do not directly assign** user input to objects in your API functions or create or update objects by directly assigning user input
- **Explicitly define** the object properties that the user is able to update in your API code
- Enforce validation and data schemas so that only **approved** object properties will be used by your API functions.

Security misconfiguration

Resource- and time-constrained engineers may use insecure default configurations for security software or appliances. Temporary configuration options used during development are commonly overlooked and make it to production, while attackers can take advantage of permissive cloud storage access policies, CORS policies, and

overly-verbose error messages that provide access to, or information about, your API to exploit it.

Avoid exposing your API to attack by properly securing it

- Ensure your deployment process is **security hardened and well documented** so that a secure hosting environment can be reproduced
- Review your deployment configurations and process regularly, including any software dependencies used in your API, deployment and configuration files, and the security of your **cloud infrastructure**
- **Limit all client interactions** with your API and any other resources (such as linked media) to secure, **authorized channels**
- Only allow API access using **necessary** HTTP verbs to reduce attack surfaces
- Set CORS policies for APIs that are **publicly accessible** from browser-based clients

Insufficient logging & monitoring

Poor application monitoring and logging allows attackers to get access to your data and infrastructure before they've been noticed, or without being noticed at all. Rigorous monitoring and accurate and informative logging are required to identify breaches and potential future threats, as well as to catch ongoing attacks before they can progress.

Your security engineers need to know about problems before they can fix them

- **Log** all authentication and authorization failures
- Log request details that can be used **to quickly identify** the source of an attack using API security tooling
- Properly format logs so that they can be filtered and reported with a **log management platform**
- Treat logs as **sensitive** data, as they contain information on both your users and API vulnerabilities
- Implement continuous monitoring of your infrastructure and tailor your monitoring reports to include the information that is **most important** to your API security.

Websocket API

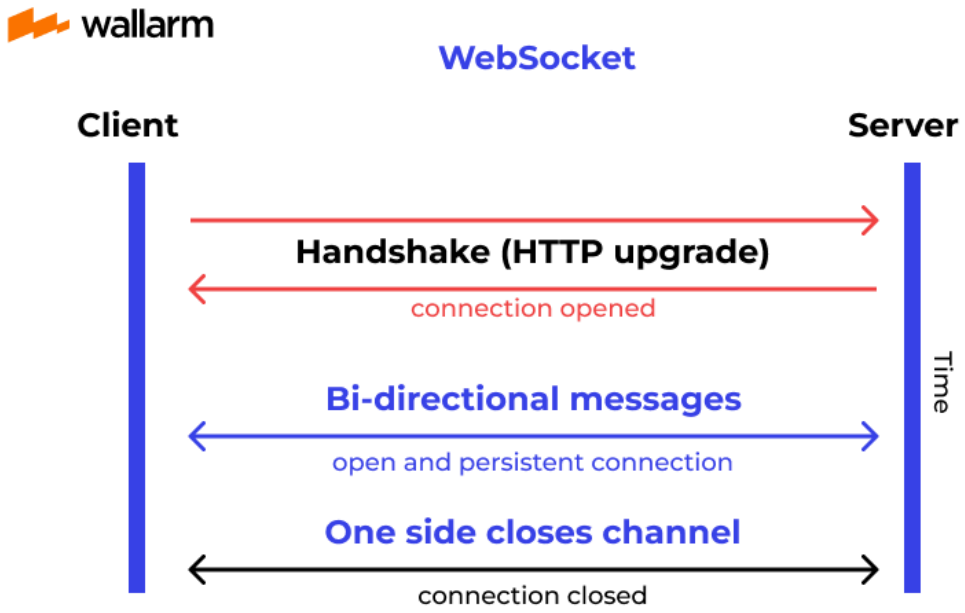
Introduction

The WebSocket API makes it possible to open a **two-way interactive communication session** between the user's browser and a server. With this API, you can send messages to a server and receive responses without having to poll the server for a reply.

The WebSocket API provides two alternative mechanisms for creating and using web socket connections: the WebSocket interface and the WebSocketStream interface.

How do webSockets work ?

As per the conventional definition, WebSocket is a **duplex protocol** used mainly in the **client-server communication channel**. It's bidirectional in nature which means communication happens to and fro between client-server.
The connection, developed using the WebSocket, lasts as long as any of the participating parties lays it off. Once one party breaks the connection, the second party won't be able to communicate as the connection breaks automatically at its front.
WebSocket need support from HTTP to **initiate the connection**. Speaking of its utility, it's the spine for modern web application development when seamless streaming of data and assorted unsynchronized traffic is concerned.



Why is a Web Socket Needed and When Should it be avoided?

WebSocket are an essential client-server communication tool and one needs to be fully aware of its utility and avoid scenarios to benefit from its utmost potential. It's explained extensively in the next section.
Use WebSocket When You Are:

- **Developing real-time web application**

  The most customary use of WebSocket is in real-time application development wherein it assists in a continual display of data at the client end. As the backend server sends back this data continuously, WebSocket allows uninterrupted pushing or transmitting this data in the already open connection. The use of WebSockets makes such data transmission quick and leverages the application's performance.
  A real-life example of such WebSocket utility is in the bitcoin trading website. Here, WebSocket assist in data handling that is impelled by the deployed backend server to the client.

- **Creating a chat application**

  Chat application developers call out WebSocket for help in operations like a one-time exchange and publishing/broadcasting the messages. As the same WebSocket connection is used for sending/receiving messages, communication becomes easy and quick.

- **Working up on gaming application**

  While gaming application development is going on, it's imperative that the server is unremittingly receiving the data, without asking for UI refresh. WebSocket accomplish this goal without disturbing the UI of the gaming app.

  Now that it's clear where WebSocket should be used, don't forget to know the cases where it should be avoided and keep yourself away from tons of operational hassles. WebSocket shouldn't be taken on board when old data fetching is the need of the hour or needs data only for one-time processing. In these cases, using HTTP protocols is a wise choice.

WebSocket vs HTTP

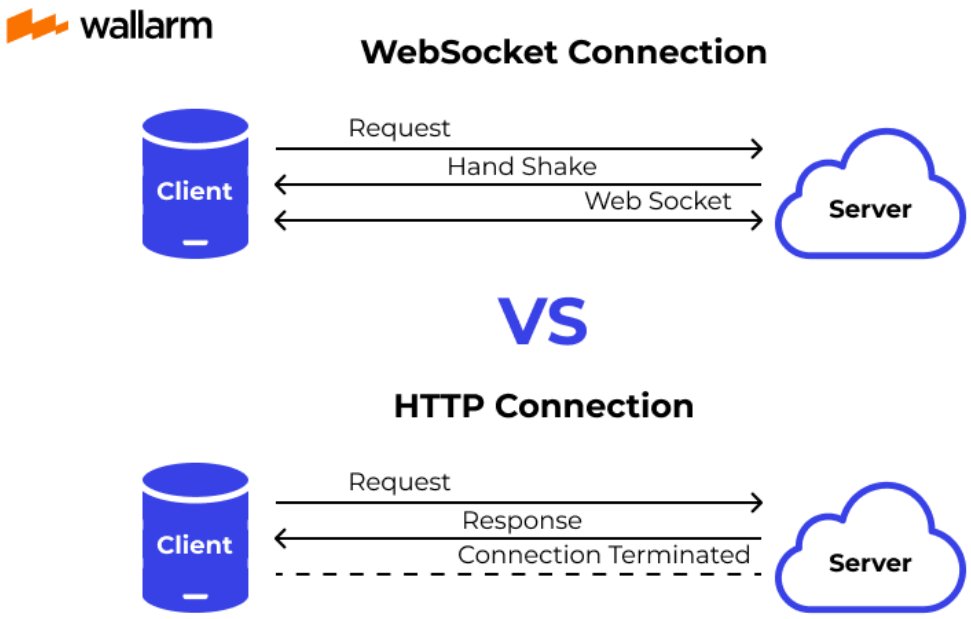
As both HTTP and WebSocket are employed for application communication, people often get confused and find it difficult to pick one out of these two. Have a look at the below-mentioned text and gain better clarity on HTTP and WebSocket.

As told previously, WebSocket is a framed and bidirectional protocol. On the contrary, to this, HTTP is a unidirectional protocol functioning above the TCP protocol.

As WebSocket protocol is capable to support continual data transmission, it's majorly used in real-time application development. HTTP is stateless and is used for the development of RESTful and SOAP applications. Soap can still use HTTP for implementation, but REST is widely spread and used.

In WebSocket, communication occurs at both ends, which makes it a **faster protocol**. In HTTP, the connection is built at one end, making it a bit sluggish than WebSocket.

WebSocket uses a **unified TCP connection** and needs one party to terminate the connection. Until it happens, the connection remains active. **HTTP** needs to **build a distinct connection** for separate requests. Once the request is completed, the connection breaks automatically.



How are WebSocket connections established?

The process starts with a WebSocket **handshake** that involves using a new scheme **ws** or **wss**. To understand quickly, you may consider them **equivalent** to **HTTP** and **secure HTTP (HTTPS)** respectively.
Using this scheme, servers and clients are expected to follow the standard WebSocket connection protocol. The WebSocket connection establishment begins with HTTP request upgrading that features a couple of headers such as Connection: Upgrade, Upgrade: WebSocket, Sec-WebSocket- Key, and so on.

The Request :

Connection: Upgrade header denotes the WebSocket handshake while the **Sec-WebSocket-Key** features **Base64-encoded random value**. This value is arbitrarily generated during every WebSocket handshake. Besides the above, the key header is also a part of t his request.

The above-listed headers, when combined,
form an HTTP GET request. It will have similar data in it:

```
GET ws://websocketexample.com:8181/ HTTP/1.1
Host: localhost:8181
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: b6gjhT32u488lpuRwKaOWs==
```

To clarify, Sec-WebSocket-Version, one can explain the WebSocket protocol version ready to use for the client.

The Response :

The response header, **Sec-WebSocket-Accept**, features the zest of value submitted in the **Sec-WebSocket-Key** request header. This is connected with a particular protocol specification and is used widely to keep misleading information at bay. In other words, it enhances the API security and stops ill-configured servers from creating blunders in the application development.
On the success of the previously-sent request, a response similar to the below-mentioned text sequence will be received:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: rG8wsswmHTJ851JgAE3M5RTmcCE=
```

Websocket Protocol

WebSocket protocol is a type of framed protocol that involves various discrete chucks with each data. It also deploys a frame type, data portion, and payload length for proper functioning. To have a detailed understanding of WebSocket protocol, knowing its building block is crucial. The foremost bits are mentioned below.

- **Fin Bit** is the fundamental bit of the WebSocket. It will be automatically generated when one begins the connection.

- **RSV1, RSV2, RSV3 Bits** are bits reserved for further opportunities.

- **Opcode** is the part of every frame and explains the process of interpreting the payload data of a particular frame. Some of the common opcode values are 0x00, 0x0, 0x02, 0x0a, 0x08, and many more.

- **Mask bit** activates when one bit is set to 1.

  WebSocket demands the use of a client-picked random key for all the payload data. Masking key, when combined with payload data, assists payload data sharing in an XOR operation. Doing so holds great importance from the application API security as masking keeps cache misinterpreting or cache poisoning at bay.

  Let's understand its crucial components in detail now:

  Payload len

  This is used for the total length encoding of the payload data in WebSocket. Payload len is displayed when the encoded data length is less than 126 bytes. Once the payload data length is exceeded from 126 bytes, additional fields are used for describing the payload length.

  Masking-key

  Every frame that the client sends to the server is masked with a 32-bit value. The masking key displays when the mask bit is 1. In the case of 0 as the mask bit, the masking key will be zero.

  Payload data

  All sorts of arbitrary application data and extension data are known as payload data. The client and servers use this data for negotiation and are used in the early WebSocket handshakes.

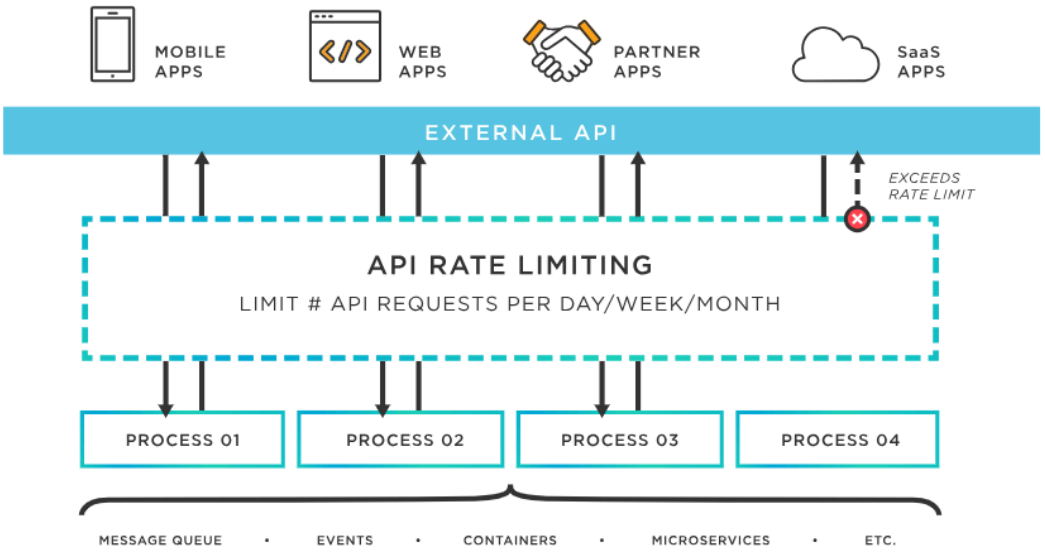
Conclusion

WebSocket enables two-way interactive communication sessions between client and server without having to poll the server for a reply, providing both speed improvements and real-capabilities over other protocols. But as with all applications, using WebSocket entails both careful programming practices and run-time protection to guard against a unique set of threats. This API defense-in-depth strategy will improve protection for both your users and your organization compared to traditional approaches.

And also don't forget about security, try the reliable solution from Wallarm - API Security Platform

API Rate Limiting and Throttling

API rate limiting is one of the fundamental aspects of managing traffic to your APIs. It is important for **quality of service**, **efficiency** and **security**. It is also one of the easiest and most efficient ways to control traffic to your APIs.



What is API rate limiting and how does it work?

An API rate limit refers to the number of calls the client (API consumer) can make in a second. Rate limits are calculated in requests per second (RPS).
Let's say you only want a client to call an API a maximum of 10 times per minute. You can apply a rate limit expressed as "10 requests per 60 seconds". The client will be able to call the API successfully up to 10 times within any 60-second interval. If they call the API any more within that time frame, they'll get an error stating they have exceeded their rate limit.

Benefits of rate limiting

API rate limiting can:

- Help with API overuse caused by accidental issues within client code, which results in the API being slammed with requests.

- Prevent a denial-of-service (DoS) attack meant to overwhelm the API resources, which could easily be executed without rate limits in place.
- Protect your API from other events that would impact its availability.
- Ensure that everyone who calls your API receives an efficient, quality service.
- Support various API monetisation models.

What are the different types of rate limiting?

There are different ways that you can approach API rate limiting.

Key-level rate limiting is focused on controlling API traffic from individual sources and making sure that users are staying within their prescribed limits. You could limit the rate of calls the user of a key can make to all available APIs (**i.e. a global limit**) or to specific, individual APIs (**a key-level-per-API limit**).

- **Global limit:** This means you restrict the total number of API calls a user can make across all services. For example, if you allow a user to make 100 calls per hour, that total applies to all APIs they access.
- **Per-API limit:** This means you can set different limits for individual APIs. For instance, a user might be allowed to make 50 calls to one API and 30 calls to another, ensuring usage is controlled based on the specific API.

API-level rate limiting assesses all traffic coming into an API from all sources and ensures that the overall rate limit is not exceeded. This limit could be calculated by something as simple as having a good idea of the maximum number of requests you could expect from users of your API. It could also be something more scientific and precise, such as the number of requests your system can handle while still performing at a high level. You can quickly establish this threshold with performance testing.

Which type of API rate limiting should you use?

These two approaches have different use cases. They can also be used in unison to power an overall API rate limiting strategy.

The simplest way to figure out which type of rate limits you should apply can be determined by asking a few questions:

- Do you want to protect against denial of service attacks or overwhelming amounts of traffic from **all users** of the API? Then, go for an **API-level global rate limit!**

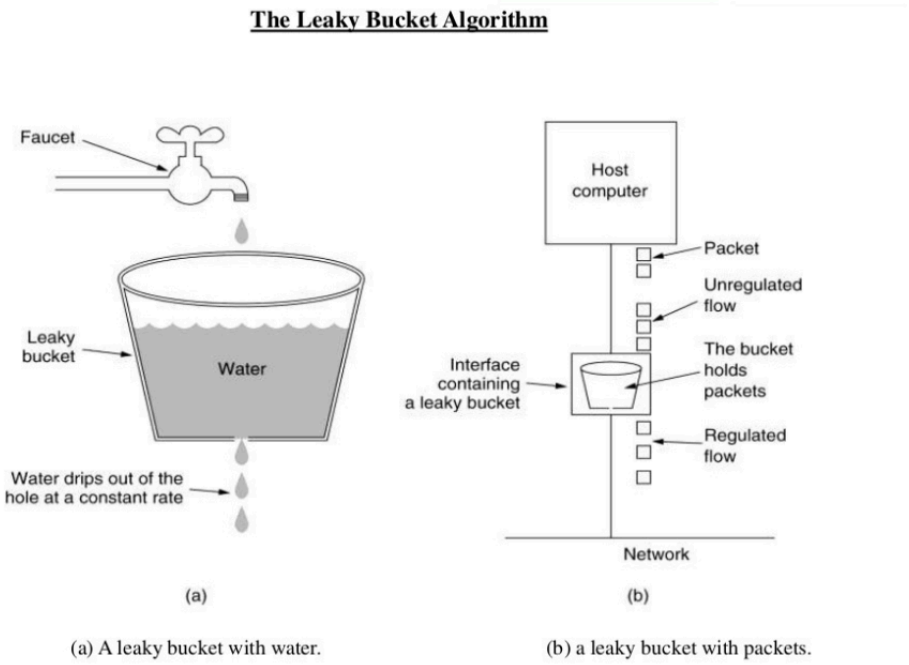
- Do you want to limit the number of API requests a specific user can make to **all APIs** they have access to? Then choose a **key-level global rate limit!**

- Do you want to limit the number of requests a specific user can make to specific APIs they have access to? Then it's time for a **key-level per-API rate limit.**

How to implement rate limiting in API environments

If you want to implement API rate limiting, you have various strategies available to you, including several algorithm-based approaches. These include:

- **Leaky bucket** – a first come, first served approach that queues items and processes them at a regular rate.



(a) A leaky bucket with water. (b) a leaky bucket with packets.

- **Fixed window** – a fixed number of requests are permitted in a fixed period of time (per second, hour, day and so on).

- **Moving/sliding window** – similar to a fixed window but with a sliding timescale, to avoid bursts of intense demand each time the window opens again.

- **Sliding log** – user logs are time stamped and the total calculated, with a limit set on the total rate.

How to test API rate limiting

It's important to test that your API rate limit is working as it should. It's not the kind of thing you want untested when you're facing a DoS attack! There are companies that will undertake API pen testing to test how robust your API security is, including how well your rate limiting works.

You'll also need to check if your API rate limits are still appropriate as your business grows. An API management tool with a handy dashboard should make it easy for you to see which limits you have in place.

How long does the rate limit last?

There is no fixed answer to how long an API rate limit lasts. It is common to apply a dynamic rate limit based on the number of requests per second, but you could also think in terms of minutes, hours or whatever time frame best suits your business model.

What is API throttling vs rate limiting?

There are two ways that requests can be handled once they exceed the prescribed limit. One is by returning an error (**via API rate limiting**); the other is by queueing the request (**though throttling**) to be executed later.

You can implement throttling at key or policy level, depending on your requirements. It's a versatile approach that can work well if you prefer not to throw an error back when a rate limit is exceeded. By using throttling, you can instead queue the request to auto-retry.

Throttling means that you can protect your API while still enabling people to use it. However, it can slow down the service that the user receives considerably, so how to throttle API requests needs careful thought in terms of maintaining service quality and availability.

What does “API rate limit exceeded” mean?

API rate limit exceeded means precisely what it says – that the client trying to call an API has exceeded its rate limit. This will result in the service producing a **429 error status response**. You can modify that response to include relevant details about why the response has been triggered.

How to bypass an API rate limit

While API rate limiting can go a long way towards protecting the availability of your APIs and downstream services, it is not without its flaws. Some individuals have worked out how to bypass an API rate limit. In fact, they’ve worked out several ways to do so.

If you use an IP-based rate-limiter, rather than key-level rate limiting, people could bypass your limits using proxy servers. They can multiply their usual quota by the number of proxies they can use.

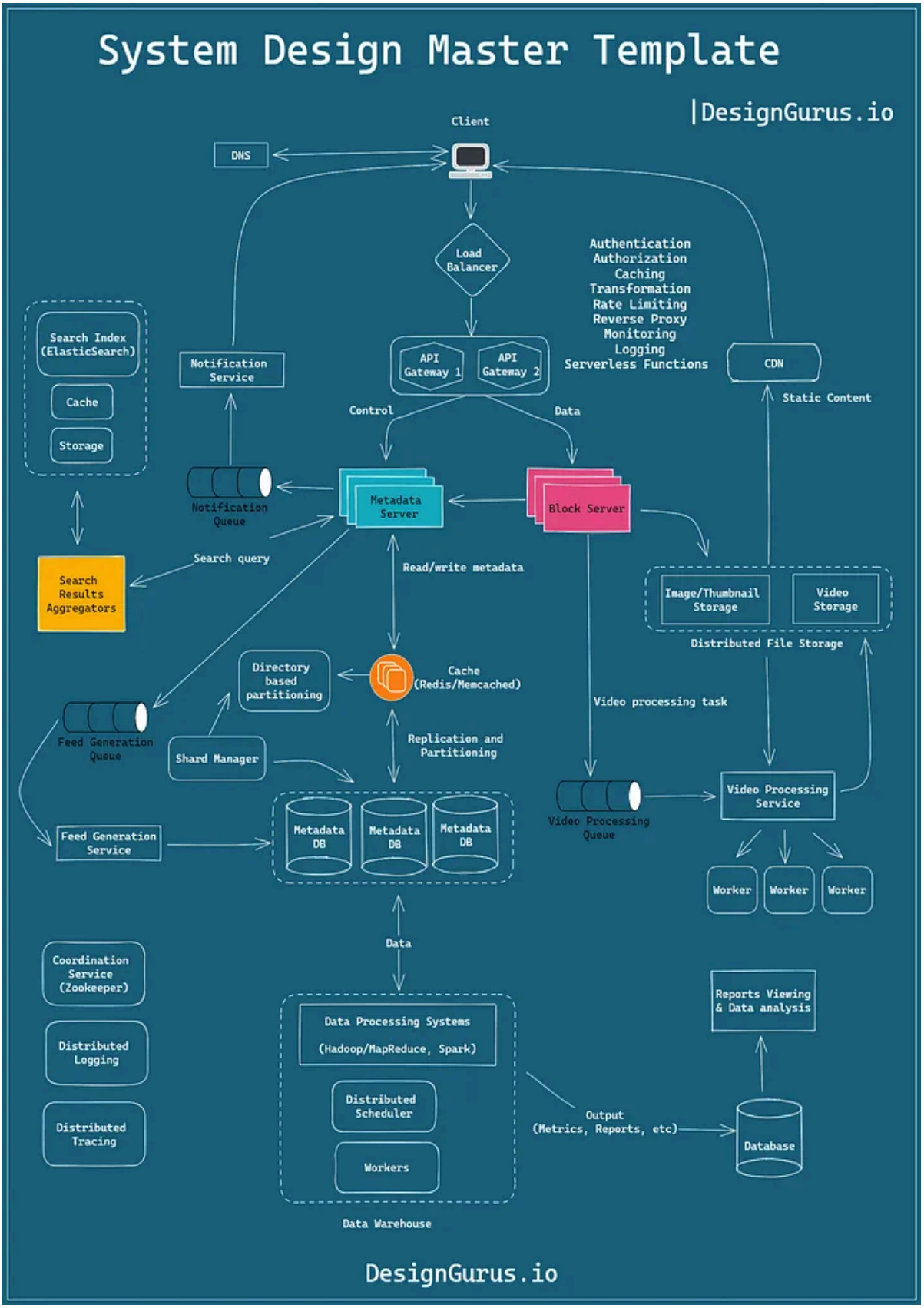
Key-based API rate limiting can also be bypassed, by people creating multiple accounts and getting numerous keys.

There are other techniques out there, such as using client-side JavaScript to bypass rate limits, so be aware that knowing how to rate limit API products doesn’t make them impervious to being bypassed!

We mentioned pen testing above. While you’re thinking about API functionality, performance and testing, why not check out this article on API testing tools?

API Monitoring

API in MicroService



API Optimizing

What is API Optimization?

API optimization refers to the process of enhancing the performance, efficiency, and overall functionality of an API. It involves a series of **techniques** and **best practices** aimed at ensuring that the API delivers data and services quickly and effectively while conserving system resources.
API optimization encompasses various aspects, including **reducing response times**, **minimizing latency**, **optimizing database queries**, **implementing caching mechanisms**, and streamlining code execution.

Developers can **enhance the user experience**, **reduce server load**, and **ensure that the API can scale to meet increasing demand**. API optimization is a crucial step in **building high-quality** and **reliable applications** that rely on APIs as a core component of their functionality.

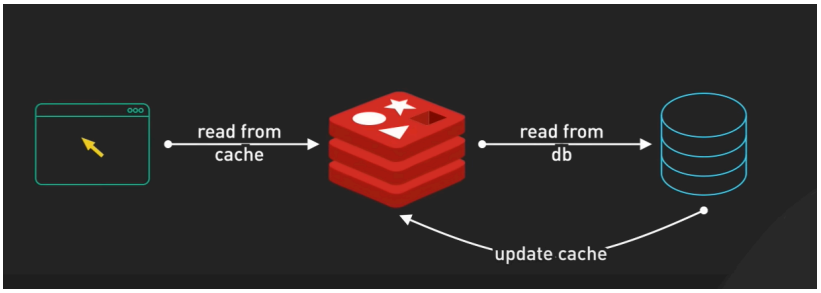
Optimization Challenges

Optimization should not be the first step in your process. Optimization is powerful, but can lead to unnecessary complexity if done prematurely.
The first step should always be to identify the actual **bottlenecks** through load testing and profiling requests .

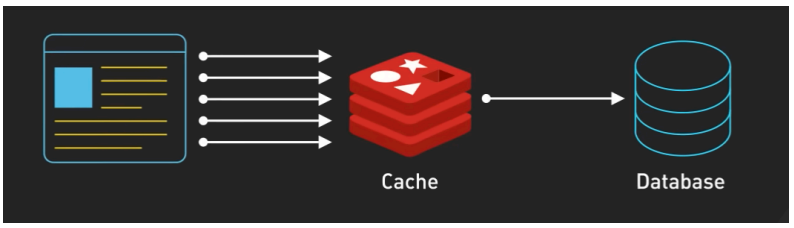
Only begin optimization once you're confirmed that an API endpoint has performance issues .

Caching

This technique is one of the most effective ways to speed up your API performance. By caching , we store the result of an expensive computation so that we can use it again later without needing to redo it later .



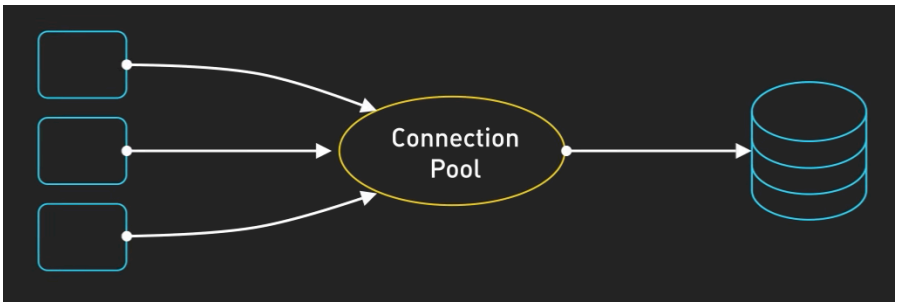
If you have an endpoint that is frequently accessed with the same request parameters, you can avoid repeated database hits by caching the response in Redis or Memcached.



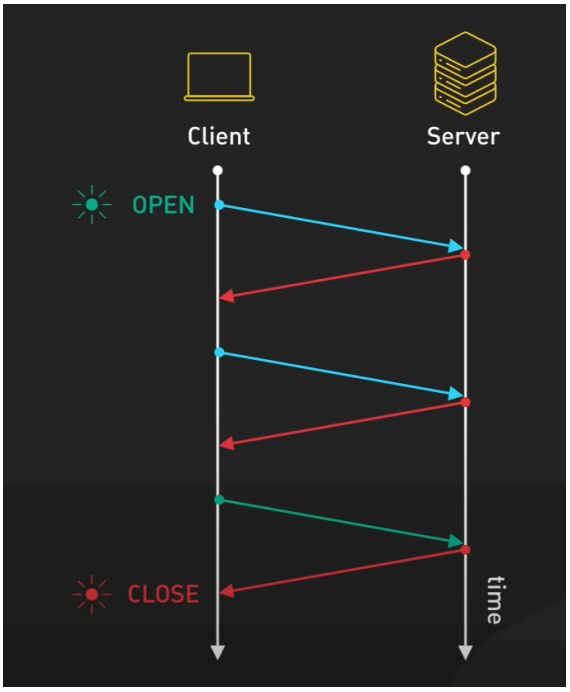
Most caching libraries make this easy to add with just a few lines of code .
Even a brief period of caching can make a significant difference in speed.

```
if redis_cache.contains(query):
   return redis_cache.get(query)
else:
   result = backend_database.query(query)
   redis_cache.put(query, result)
   return result
```

Connection Pool + (useful)

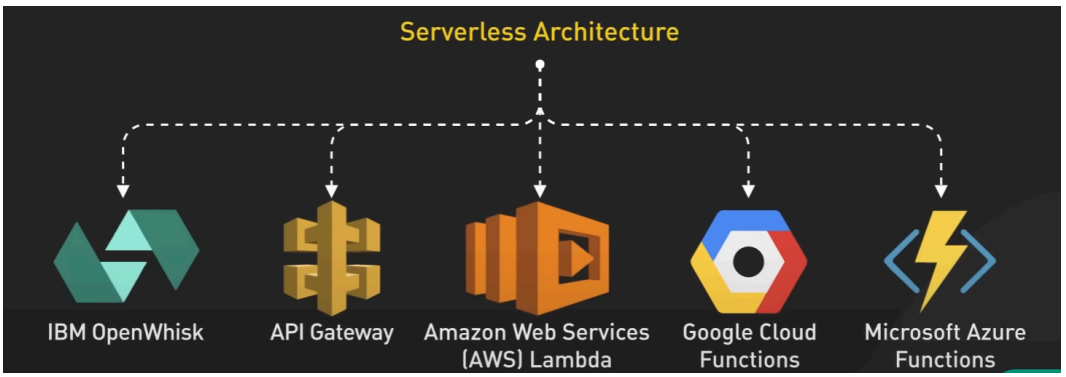


This optimization technique involves maintaining a pool of open connections, rather than opening a new database connection for each API call.
Creating a new connection each time involves a lot of handshake protocols and setup which can slow down your API .
This reuse of connections can greatly improve throughput.

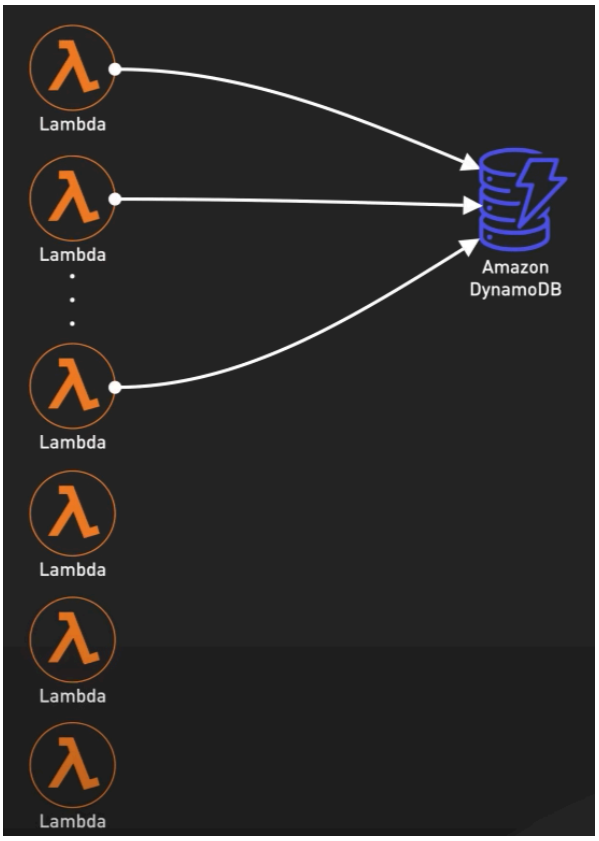


Classic Technique Pool Connection

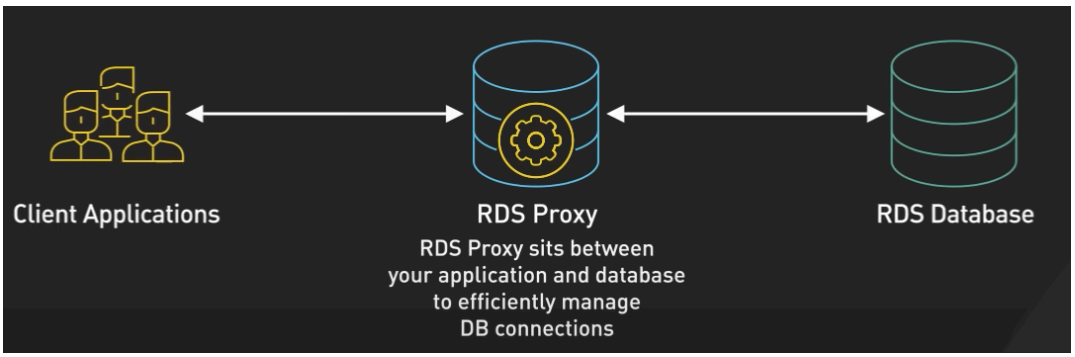
If you are using a **serverless architecture**, connection management can be a bit more challenging .



This is because each serverless function instance typically opens its own database connection, and because serverless can scale rapidly, this could potentially lead to a large number of open connections that can overwhelm the database.



Solutions like AWS RDS Proxy and Azure SQL Database serverless are designed to handle this situation and manage a connection pooling for you.



Connection pooling keeps a group of open database connections ready to be used, so your application can quickly borrow a connection when it needs to access the database, just like a restaurant keeps tables ready for customers to sit at. This makes the whole process quicker and more efficient!

Avoid N+ Problem +(a lot of query problem)

What is the N+1 query problem?

The N+1 query problem happens when your code asks the database for one item and then, for each of those items, it asks for more information.
For example:

1. You first get a list of 10 items (that's 1 query).
2. Then, for each of those 10 items, you ask for extra details (that's 10 more queries).

So, in total, you end up with 11 queries (1 + 10 = N+1). This can slow things down a lot.
How to avoid it:

1. **Use Joins:** Instead of asking for each item separately, ask for all the information you need in one go using a single query with joins.
2. **Eager Loading:** If you're using an ORM (Object-Relational Mapper), use eager loading to fetch related data at the same time as the main data.
3. **Batch Queries:** If you can't avoid multiple queries, try to batch them together so you ask for multiple items in one query.

By doing these, you can reduce the number of queries and make your app faster!

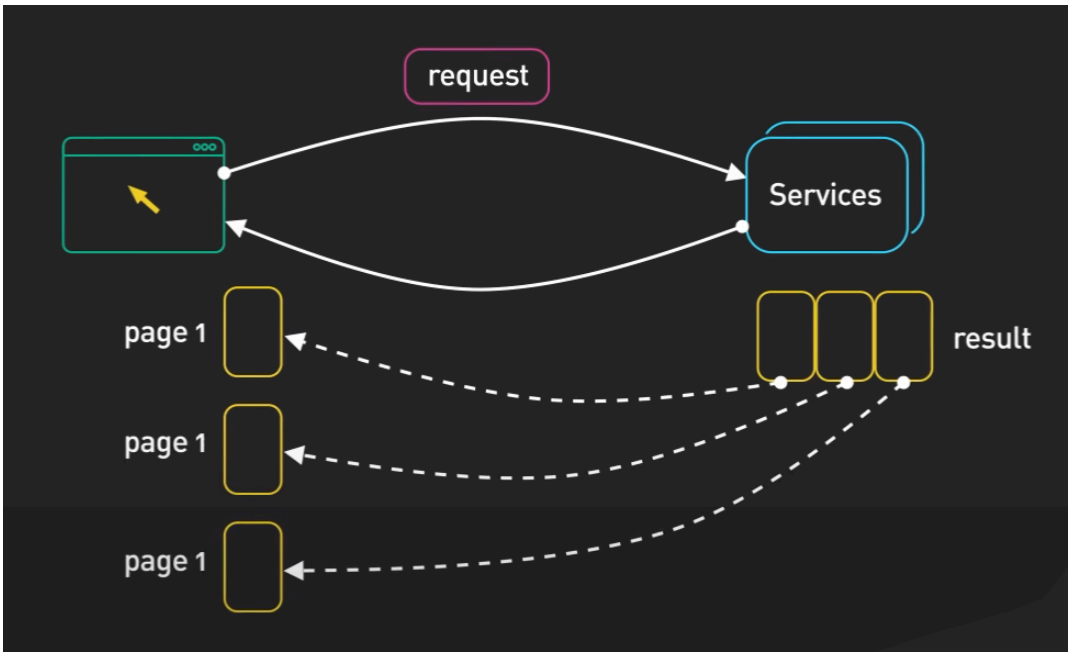
Conclusion?

The N+1 query problem arises when your code makes one initial query plus additional queries for each item retrieved, leading to multiple database requests that can slow down your application. To avoid this issue, combine your queries into a single, more efficient request using methods like JOINs or eager loading. By doing

so, you reduce the number of database trips and improve performance, allowing your application to run faster and more efficiently. Always aim for fewer, larger queries instead of multiple small ones for better overall efficiency!

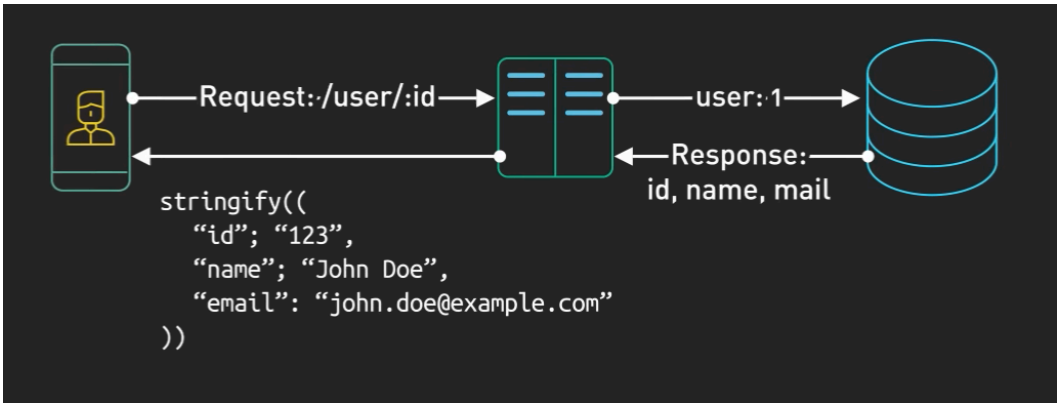
Pagination

If your API response returns a large amount of data, it can slow things down. instead, break the response into smaller more manageable pages using limit and offset parameters. This can speed up data transfer and reduce load on the client side.

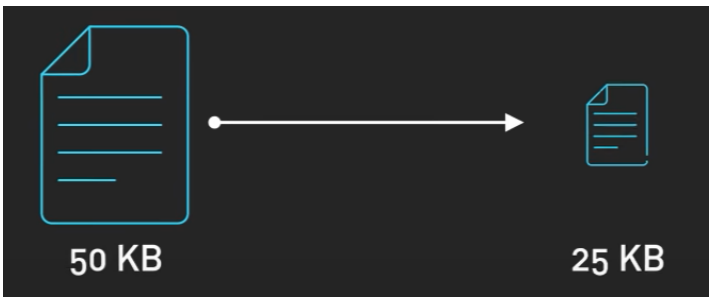


JSON Serializers + (don’t needed)

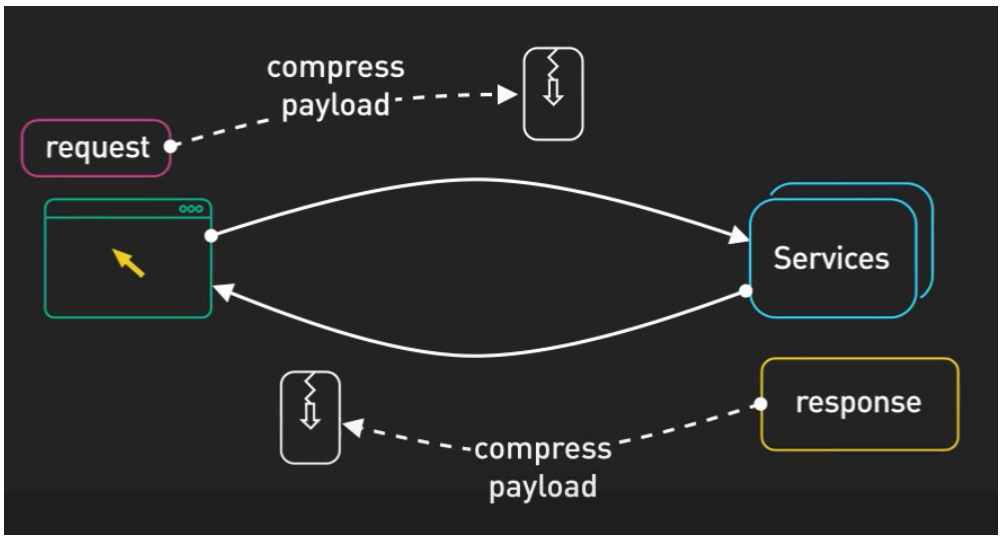
When returning JSON responses from your API, the speed of your serialization process can make a noticeable difference in response times. Consider using a fast serialization library to minimize the time spent converting your data into JSON format.



Payload Compression + (compress the ressources)



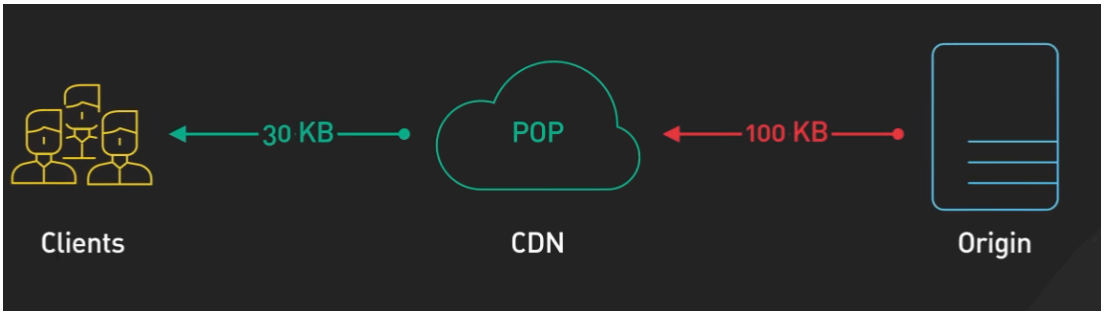
By enabling compression on large API response payloads, you can reduce the amount of data transferred over the network. The client then decompresses the data.



There are even more efficient algorithms like Brotli that provide better compression ratios.

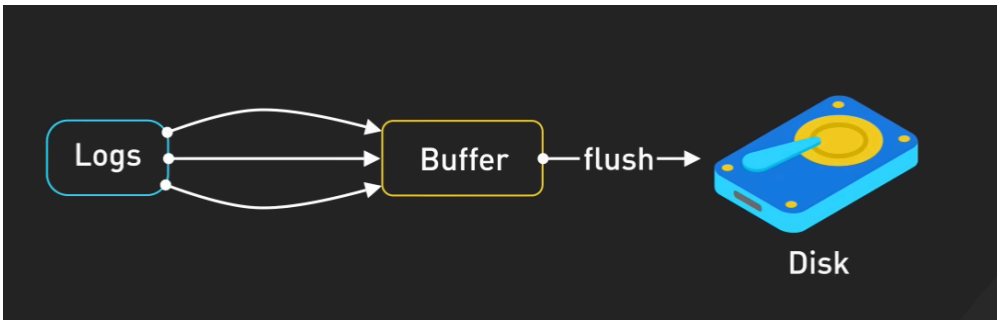
+

Also, many Content Delivery Networks (**CDNs**) like Cloudflare can handle compression for you,offloading this task from your server



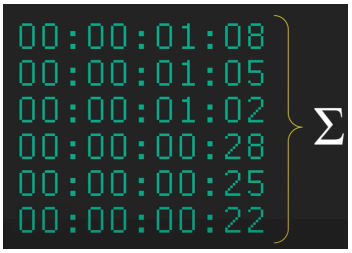
+

Asynchronous Logging + (helps)

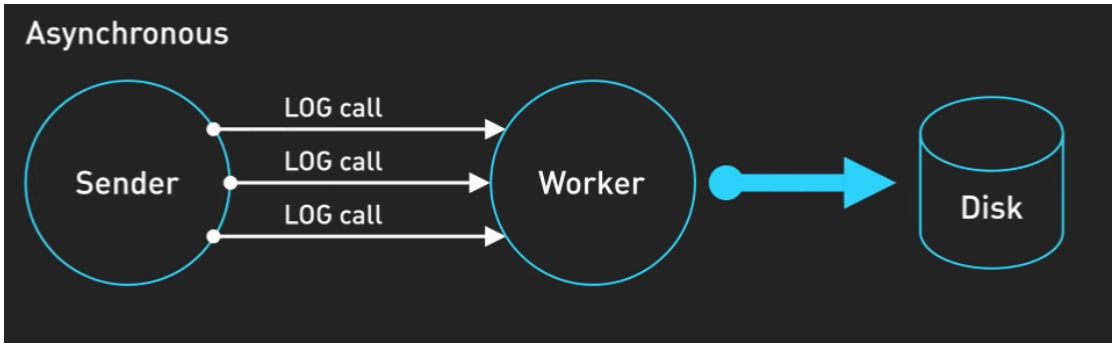


In many applications, the time it takes to write logs is negligible.
However, in high-throughput systems where every millisecond counts, the time taken to write logs can add up

.



In such cases, asynchronous logging can help.
This involves the main application thread quickly buffering, while a separate logging thread writes the log entries to the file or sends them to the logging service.



Just keep in mind that with asynchronous logging, there's a small chance you might lose some logs if your application crashes before the logs have been written.

Use Ranges or Rate limiting + (not useful)

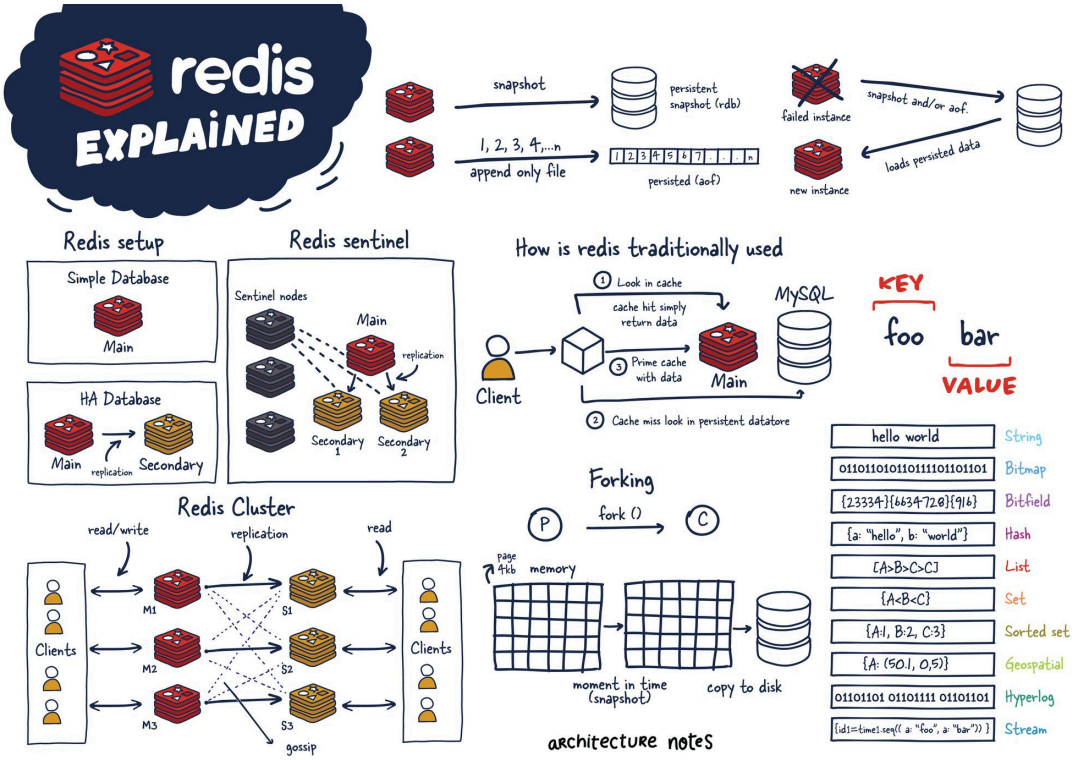
Restrict results by giving structure created by users. You can define the start and end to provide only elements available within those ranges. This optimization allows you to limit the response package. Using this technique allows for the processing of data on the server rather than the client. Setting ranges helps reduce the instance of receiving unwanted or unusable data while also reducing the load on the API, which provides better results.

Filtering (not useful)

An often overlooked tactic, filtering plays an important part in optimizing an API. If you want to limit the results of parameters from a requester, you can use filtering. You can identify which resources are given to the user and can provide measurable optimization. You also get a better overall user [experience] providing only the request issues and let the entries process run better.

Redis

Redis is a powerful, open-source, in-memory data structure store widely used for caching, real-time analytics, message brokering, and more. Its high performance, versatility, and rich feature set make it an excellent choice for optimizing APIs. Below is an in-depth overview of Redis, its features, use cases, and best practices for leveraging it effectively in API optimization.



What is Redis?

Redis (Remote Dictionary Server) is an open-source, in-memory data store that can be used as a database, cache, and message broker. It supports various data structures such as strings, hashes, lists, sets, sorted sets, bitmaps, hyperloglogs, and geospatial indexes with radius queries.

Key Characteristics:

- **In-Memory Storage:** Data is stored in RAM, enabling extremely fast read and write operations.
- **Persistence Options:** Although primarily in-memory, Redis offers persistence through snapshotting (RDB) and append-only files (AOF) to ensure data durability.
- **Rich Data Structures:** Supports a variety of data types beyond simple key-value pairs, allowing complex data manipulation.
- **Atomic Operations:** All Redis operations are atomic, ensuring data consistency.
- **Pub/Sub Messaging:** Supports publish/subscribe messaging paradigms for real-time communication.
- **Replication and High Availability:** Provides master-slave replication, automatic failover with Redis Sentinel, and clustering for scalability.

Key Features of Redis

1. Data Structures:

Redis supports a variety of data structures that cater to different use cases:

- **Strings:** Simple key-value pairs, suitable for caching individual items.
- **Hashes:** Useful for storing objects with multiple fields, such as user profiles.
- **Lists:** Ordered collections of items, ideal for implementing queues or stacks.
- **Sets:** Unordered collections of unique items, useful for membership checks.
- **Sorted Sets:** Similar to sets but with ordering based on scores, suitable for ranking systems.
- **Bitmaps, HyperLogLogs, Geospatial Indexes:** Advanced data structures for specialized applications.

2. Persistence:

Redis provides multiple persistence mechanisms to balance between performance and data durability:

- **RDB (Redis Database Backup):** Creates point-in-time snapshots of the dataset at specified intervals.
- **AOF (Append-Only File):** Logs every write operation received by the server, allowing for more fine-grained recovery.
- **Hybrid Approach:** Combining RDB and AOF for both performance and durability.

3. Replication and High Availability:

- **Master-Slave Replication:** Allows data to be copied from a master node to one or more slave nodes.
- **Redis Sentinel:** Provides high availability by monitoring Redis instances, performing automatic failover, and notifying clients of topology changes.
- **Redis Cluster:** Enables horizontal scaling by partitioning data across multiple Redis nodes, supporting large datasets and high throughput.

4. Performance:

- **Low Latency:** Operations typically execute in sub-millisecond times due to in-memory storage.
- **High Throughput:** Capable of handling millions of requests per second with proper configuration and hardware.

5. Extensibility:

- **Lua Scripting:** Allows for executing custom scripts atomically within Redis.
- **Modules:** Extend Redis functionality with custom commands and data types (e.g., RedisJSON, RediSearch).

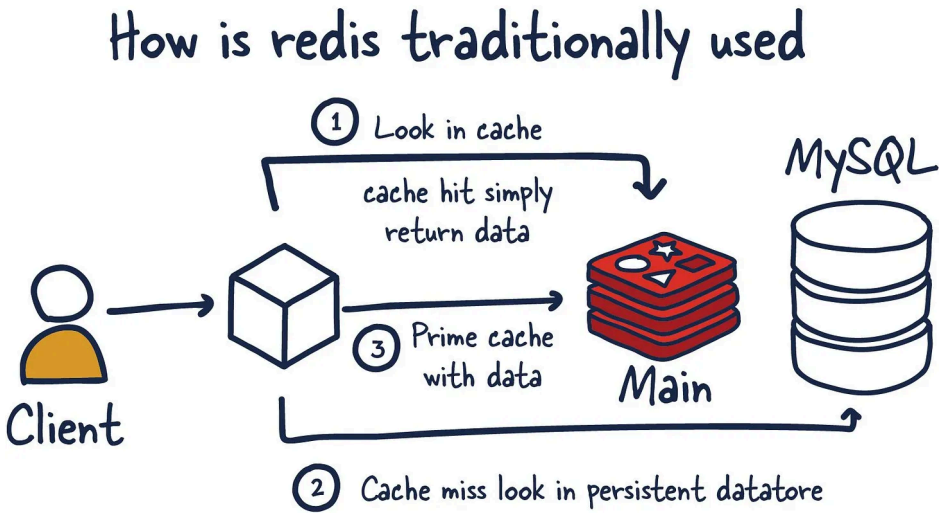
Use Cases for Redis in API Optimization

1. Caching:

Redis is predominantly used as a caching layer to store frequently accessed data, reducing the load on primary databases and decreasing response times.

- **Example:** Caching API responses for endpoints that fetch user profiles, product details, or configuration settings.
- **Benefits:**
    - **Reduced Latency:** In-memory access significantly speeds up data retrieval.
    - **Lower Database Load:** Decreases the number of read operations hitting the primary database.

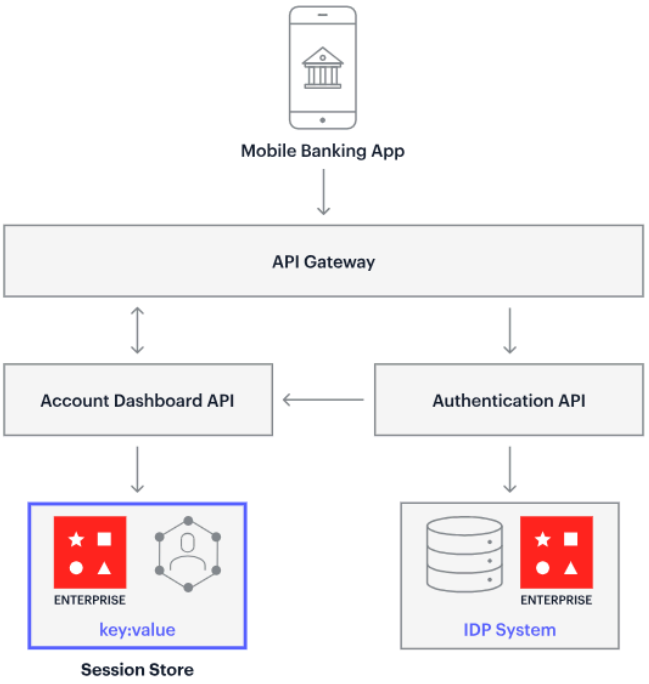
- **Scalability:** Supports high read/write throughput necessary for high-traffic APIs.



2. Session Management:

Storing user sessions in Redis allows for fast access and scalability, especially in distributed systems.

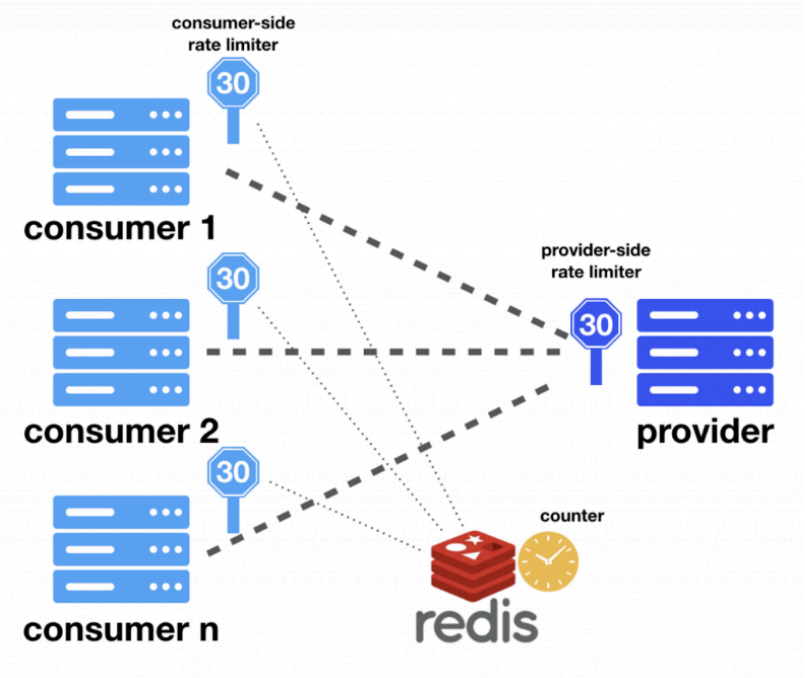
- **Example:** Managing user authentication tokens or session data in a stateless API architecture.



3. Rate Limiting:

Implementing rate limiting to control the number of API requests per user or IP address within a specific timeframe.

- **Example:** Using Redis atomic increment operations with expiration to track and enforce request limits.



4. Real-Time Analytics:

Processing and storing real-time metrics, such as API usage statistics, in Redis for immediate analysis and visualization.

- **Example:** Tracking the number of API calls per endpoint per minute.

5. Message Brokering:

Using Redis' Pub/Sub capabilities to handle real-time messaging between different parts of your application.

- **Example:** Notifying services of events or changes that occur within the API.

6. Distributed Locks:

Managing distributed locks to ensure that critical sections of code are executed by only one instance in a distributed environment.

- **Example:** Preventing race conditions when multiple API instances attempt to modify shared resources.

API Versioning

What is API versioning?

API versioning is the process of **managing** and **tracking changes** to an API. It also involves **communicating** those changes to the **API's consumers**.

Change is a natural part of API development. Sometimes, developers have to update their API's code to fix security vulnerabilities, while other changes introduce new features or functionality. Some changes do not affect consumers at all, while others, which are known as "**breaking changes**," lead to **backward-compatibility issues**, such as unexpected errors and data corruption. **API versioning ensures that these changes** are **rolled out successfully** in order to preserve consumer **trust** while **keeping the API secure**, **bug-free**, and **highly performant.**

Here, we'll review the benefits of API versioning and discuss several scenarios in which it is necessary. We'll also explore some of the most common approaches to API versioning, provide five steps for successfully versioning an API, and highlight some best practices for API versioning. Finally, we'll discuss how the Postman API Platform can support your API versioning workflow.

What are the benefits of API versioning?

It's essential for an API's producers and consumers to stay in sync as the API evolves—regardless of whether it is private or public. An effective API versioning strategy not only enables API producers to iterate in a way that minimizes the consumer-facing impact of breaking changes, but also provides a framework for effectively communicating these changes to consumers. This transparency builds trust and—in the case of public APIs—strengthens the organization's reputation, which can boost the API's adoption and retention rates.

When should you version an API?

You should version your API whenever you make a change that will require consumers to modify their codebase in order to continue using the API. This type of change is known as a "breaking change," and it can be made to an API's input and

output data structures, success and error feedback, and security mechanisms. Some common examples of breaking changes include:

- **Renaming a property or endpoint**: You might sometimes want to rename a property or method so that its meaning is clearer. While clear naming is important from an API design standpoint, it's almost impossible to change property or method names once the API is in production without breaking your consumers' code.

- **Turning an optional parameter into a required parameter:** As your API evolves, you may notice instances in which a certain input parameter should be mandatory, even though it was initially designed to be optional. While this type of change may help standardize inputs and make API operations more predictable, it will result in missing property errors for clients that are not programmed to pass a value for this property.

- **Modifying a data format or type:** You may sometimes realize that several properties, such as firstName and lastName, should instead exist within a user object, instead of as separate properties that take string values. While this type of change would improve your API's design, it is nevertheless a breaking change that will cause a parsing exception.

- **Modifying a property's characteristics:** You may occasionally be tempted to change the rules for certain properties. For instance, a description property of type: string may have a maxLength rule that you discover is too low or too high. This type of change will have different results depending on its implementation, including database and UI errors.

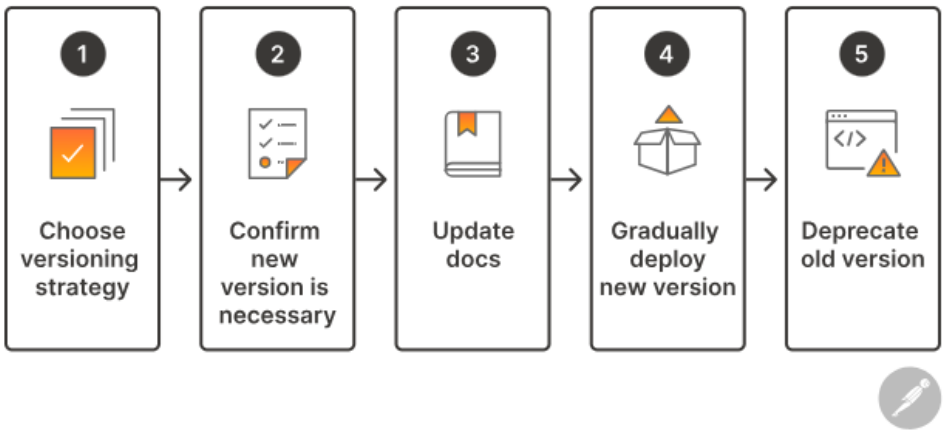
What are some types of API versioning?

There are several approaches to API versioning, including:

- **URL versioning:** With this approach, the version number is included in the URL of the API endpoint. For instance, consumers who are interested in viewing all of the products in a database would send a request to the `https://example-api.com/v1/products` endpoint. This is the most popular type of API versioning.
- **Query parameter versioning:** This strategy requires users to include the version number as a query parameter in the API request. For instance, they might send a request to `https://example-api.com/products?version=v1`.

- **Header versioning**: This approach allows consumers to pass the version number as a header in the API request, which decouples the API version from the URL structure.
- **Consumer-based versioning:** This versioning strategy allows consumers to choose the appropriate version based on their needs. With this approach, the version that exists at the time of the consumer's first call is stored with the consumer's information. Every future call is then executed against this same version—unless the consumer explicitly modifies their configuration.

It's important to note that these versioning strategies are used in tandem with a versioning scheme, such as semantic versioning or date-based versioning. Semantic versioning follows a three-part number format (i.e., 3.2.1), in which the first number represents a major update that might include breaking changes, the second number represents an update that includes new, backward-compatible features, and the third number represents bug fixes or patches. Date-based versioning, in contrast, identifies versions with the specific date on which they were released.



How do you version an API?

API versioning directly affects the overall success of an API, and it requires careful planning to ensure it is executed in a methodical way. API producers should follow these steps to version their API as effectively as possible:

Step 1: Choose a versioning strategy

It's important to choose an API versioning strategy during the API design phase of the API lifecycle. This versioning strategy should be shared across all APIs in your portfolio. The earlier you think about versioning, the more likely you are to choose resilient design patterns that will reduce the occurrence of breaking changes. An early decision about API versioning will also help your team align on a realistic roadmap for how your APIs will evolve to meet consumer needs for years to come.

Step 2: Confirm whether a new version is necessary

Change is an inevitable part of API development, but not every change necessitates a new version. Before deciding to roll out a new version, teams should assess the scope and impact of the change they want to make—and determine if there is a way to make it in a backward-compatible way. For instance, you may opt to add a new operation instead of modifying an existing one. If there's no way to avoid a breaking change, you might consider waiting to introduce it until you release an exciting new feature that will improve your consumers' experience.

Step 3: Update the documentation

Interview Questions For RESTful

Beginner Questions

How are REST APIs stateless?

to answer this question we need to First understand the meaning of statefulness and statelessness
an application , API or web service that is considered stateful stores data from the client on its own servers for example if a username and password were passed from the client to the server as a form of authentication and the server stores that data then the web server is stateful this is because the server is now storing data from the client on the other hand the rest architecture requires that the client states is not stored on the server instead each request made by the client must contain all necessary information for that particular HTTP method so that's how rest APIs are stateless

Explain the HTTP methods
alright on to question number two explain the HTTP methods restful web services will use HTTP methods and client requests the most common methods are get post put and delete get fetch is a resource from the server post requests for a resource to be created on the server put requests for a resource to be updated delete requests for a resource to be deleted these four methods correspond to crud operations or create read update and delete

Explain the HTTP codes
restful web services use HTTP status codes in server responses the most common types of status codes are 200 which represent a successful request and response 400 codes represent a client-side error and 500 codes represent a server-side error

What is a URI?
URI stands for uniform resource identifier it identifies every resource in the rest architecture a URI can be one of two types URN which identifies a resource through a unique and persistent name like a books ISBN and URL which is your typical web address on the web URLs will typically be used when designing rest APIs

Best practices in making URI for RESTful web services?

practices in making the URI for a restful web service Uris should be mostly standardized When developing a restful web service this way clients can more easily work with the web service here are five best practices when making restful Uris develop them with the understanding that forward slashes indicate hierarchy use plural nouns for branches use hyphens for multiple words use
lowercase and refrain from using file extensions

Differences between REST and SOAP?

well rest is an architecture used to develop web services soap or simple object access protocol serves as a protocol for exchanging structured information by way of apis while rest has flexible standards soap standards are much more strict in that their implementation and statefulness often means that the client and server are closer connected additionally while rest allows for data transfer in Json XML and others soap only supports XML in general soap is a stricter and more Niche alternative to rest it's used in cases where more regulated and staple data needs to be transferred

Differences between REST and AJAX?
differences between rest and Ajax Ajax refers to asynchronous JavaScript and XML is a collection of web technologies that allow for asynchronous web applications using the built-in XML HTTP request object while rest API refers to an architecture for handling HTTP requests Ajax refers to a collection of web Technologies for making asynchronous web requests this means that a rest API may handle Ajax clients and that Ajax may be used to send restful requests but a rest API could never be implemented nor replaced by Ajax

Real-world examples of REST APIs?
REST apis can be seen in nearly every facet of the web by going to the exponent website there is a get request sent to a server at the URL for the necessary HTML to display to the client most commonly as a developer rest apis are used to manipulate data from a particular database using the four main HTTP methods operations like retrieving file data accessing images and even hosting a website all require use of rest apis

Advanced Questions

What is the difference between PUT, POST, and PATCH?

PUT POST and PATCH are very common http methods the key difference that PUT used to update a specified existing resource
while POST used to create a new resource there's also PATCH that is another way to update resource the diff between PUT and PATCH that PATCH alters only the data that is specified where PUT replace the entire resource with a new resource comprised with the data

What is a payload in the context of a REST API?

payload refers to any data that has been transferred via request or response through the rest api for instance if the client makes a get request the server will provide a response with a payload or the client will provide the payload like any post request where the client giving the data to the server

What is a REST message?

the concept of messaging refers to back and forth communication via request and response by the client and server there is typically a message with every response body associated with provided status code this message is one of the most important parts of debugging and testing a rest api example of a message include confirmation of resource was created or that any exceptions were raised

What are the core components of an HTTP request?

- **Method** :or the verb refers to operations like get post, put, delete etc..
- **URI**:this used to identify the resource typically using a url
- **HTTP version** :this indicate the http version that is being used typically http 1.1 or http 2.0
- **Request Header** : contains the metadata which can be the message format cache settings content format etc..
- **Request Body** :contain the content or data that being sent

What are the core components of an HTTP response?

- **Status code** : provide info about the success or failure of a request
- **Http version** : this indicate the http version that is being used typically http 1.1 or http 2.0
- **Response Header** : contain response meta data like content length or content type date etc..
- **Response Body** : contain the content or the return data

What is an idempotent method and why are they important?

when using restful web services a client may make a numerous requests to a server at a time an idempotent is one that yields the same response regardless of how many times a request is sent for example imagine pressing a button on a webpage that make a get request to a resource after (n)th press button the api response will still be the same as the first press
rest apis are designed on the principle that the client should not depend upon previous api calls hence why the rest architecture is made to be stateless therefore res apis should be made such their methods are idempotent when possible

What's the difference between idempotent and safe HTTP methods?

idempotent method is one that can be called multiple time without changing responses

safe http method is one that does not change a resource , meaning it may read but not right like the get method its consider a safe method

Explain caching in a RESTful architecture?

a client may request the same data from the rest api multiple time in this case it's been official for the response to be cached bandwidth is needed and the client retrieve data faster each rest api contains specific metadata related to the caching of responses for example header of cache control and expires specify what responses may be cached by whom and for how long

Resources

Ressource (1)
What is an API?

JWT
ازاي احمي الموقع بتاعي ؟ (JSON Web Tokens - JWT)

Ressource (2)

https://blog.hubspot.com.

Ressource (3)

https://www.redhat.com

Ressource (4) REST

https://www.redhat.com/en/resources

Ressource (5) URI and URL

https://www.hostinger.fr/tutoriels

Ressource (6) API auth

https://www.postman.com/api-platform/api-authentication.

Ressource (7) Rate limiting

https://tyk.io/learning-center/api-rate-limiting/

Ressource (8) Websocket

https://www.wallarm.com/what/a-simple-explanation-of-what-a-websocket-is#

Ressource (9) Rate limiting By OpenIA

https://platform.openai.com/docs/guides/rate-limits/usage-tiers?context=tier-free

Ressource (10) API Optimize 7 Techniques

Top 7 Ways to 10x Your API Performance

Ressource (11) Tips for Optimizing an API | Dreamfactory

https://blog.dreamfactory.com/8-tips-for-optimizing-an-api

Ressource (12) Redis Architecture

https://architecturenotes.co/p/redis