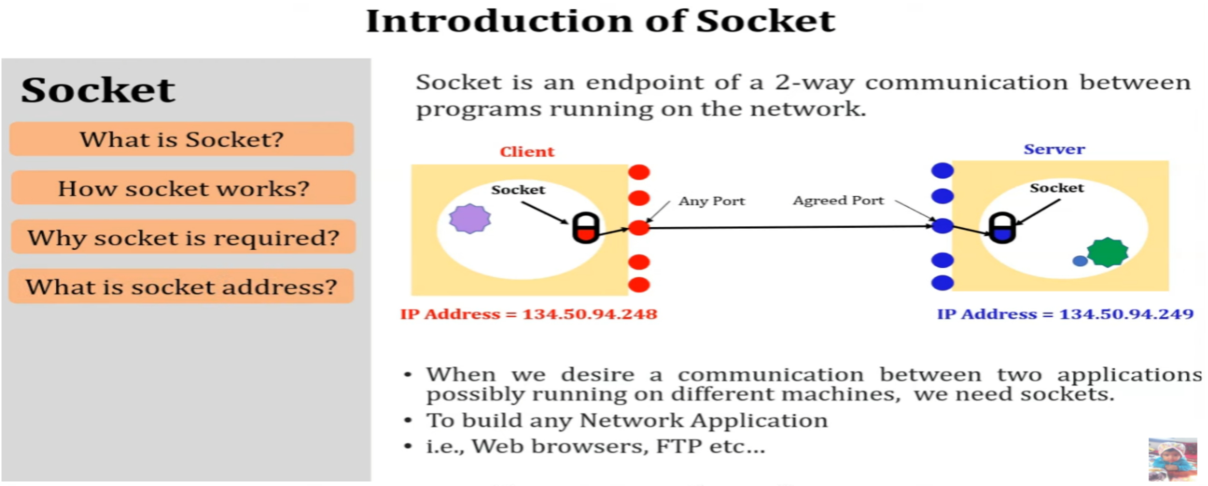
An **API (Application Programming Interface)** is a set of rules and protocols that allow different software applications to communicate with each other. It defines how requests and responses should be structured, enabling seamless interaction between systems.

Types of APIs:

1. **Web APIs** – Used to communicate over the internet (e.g., REST, GraphQL).
2. **Library APIs** – Provide functionality within a programming language (e.g., TensorFlow API).
3. **Operating System APIs** – Allow apps to interact with system resources (e.g., Windows API).
4. **Hardware APIs** – Enable software to control hardware (e.g., OpenGL for graphics).



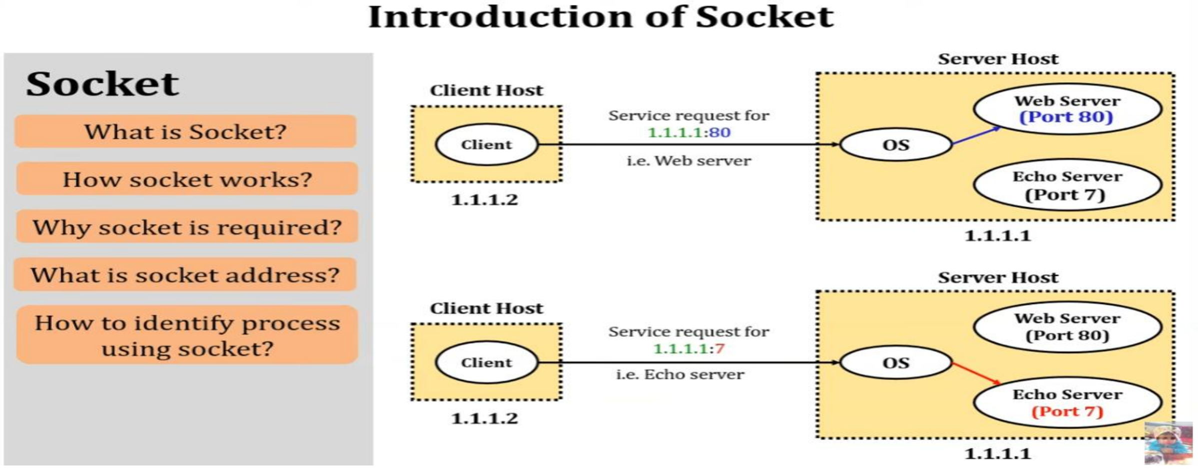
Introduction of Socket

Socket

- What is Socket?
- How socket works?
- Why socket is required?
- What is socket address?

Socket is an endpoint of a 2-way communication between programs running on the network.

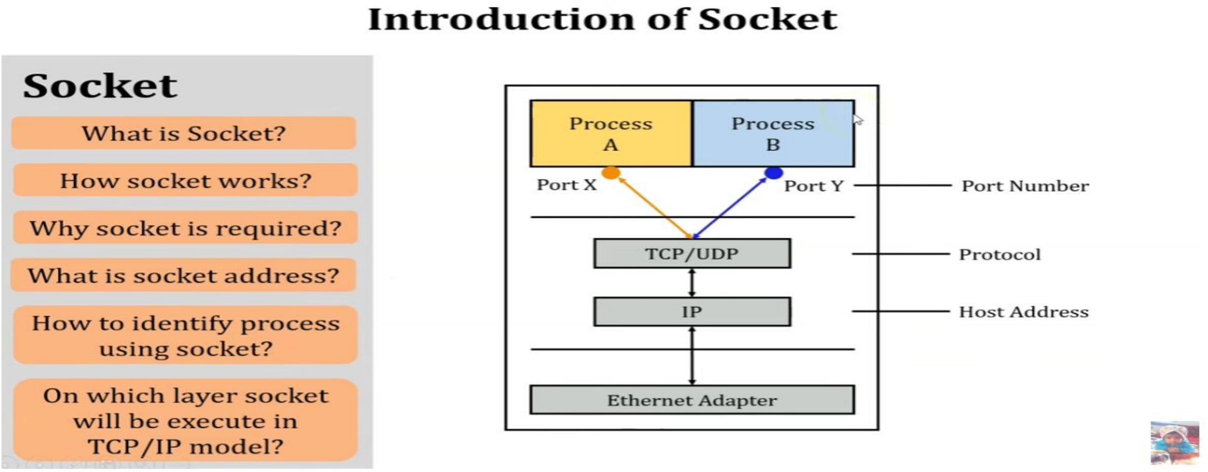
- When we desire a communication between two applications possibly running on different machines, we need sockets.
- To build any Network Application
- i.e., Web browsers, FTP etc...



Introduction of Socket

Socket

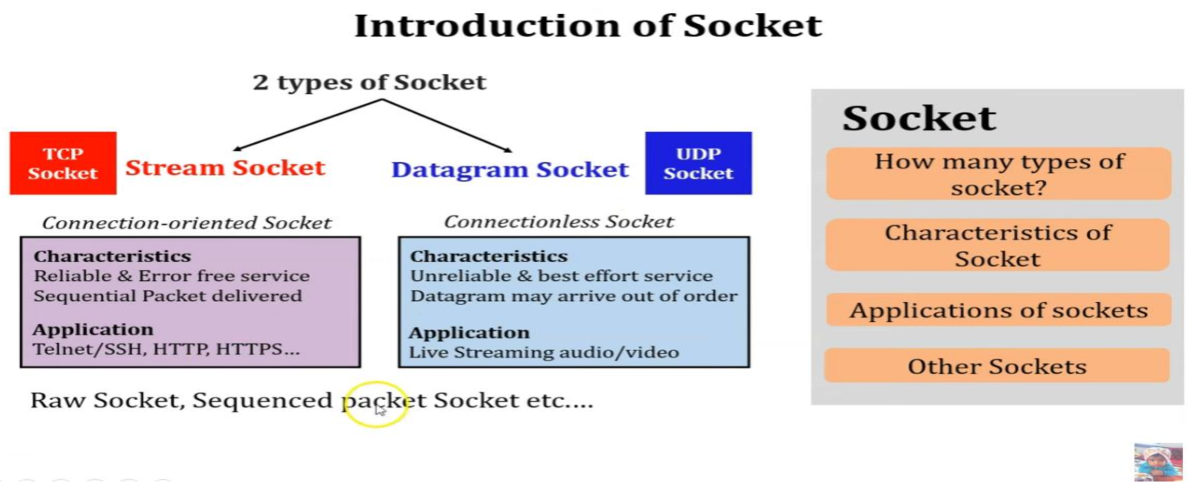
- What is Socket?
- How socket works?
- Why socket is required?
- What is socket address?
- How to identify process using socket?



Introduction of Socket

Socket

- What is Socket?
- How socket works?
- Why socket is required?
- What is socket address?
- How to identify process using socket?
- On which layer socket will be execute in TCP/IP model?



Introduction of Socket

2 types of Socket

TCP Socket	Stream Socket	Datagram Socket	UDP Socket

Connection-oriented Socket

Characteristics
Reliable & Error free service
Sequential Packet delivered

Application
Telnet/SSH, HTTP, HTTPS...

Connectionless Socket

Characteristics
Unreliable & best effort service
Datagram may arrive out of order

Application
Live Streaming audio/video

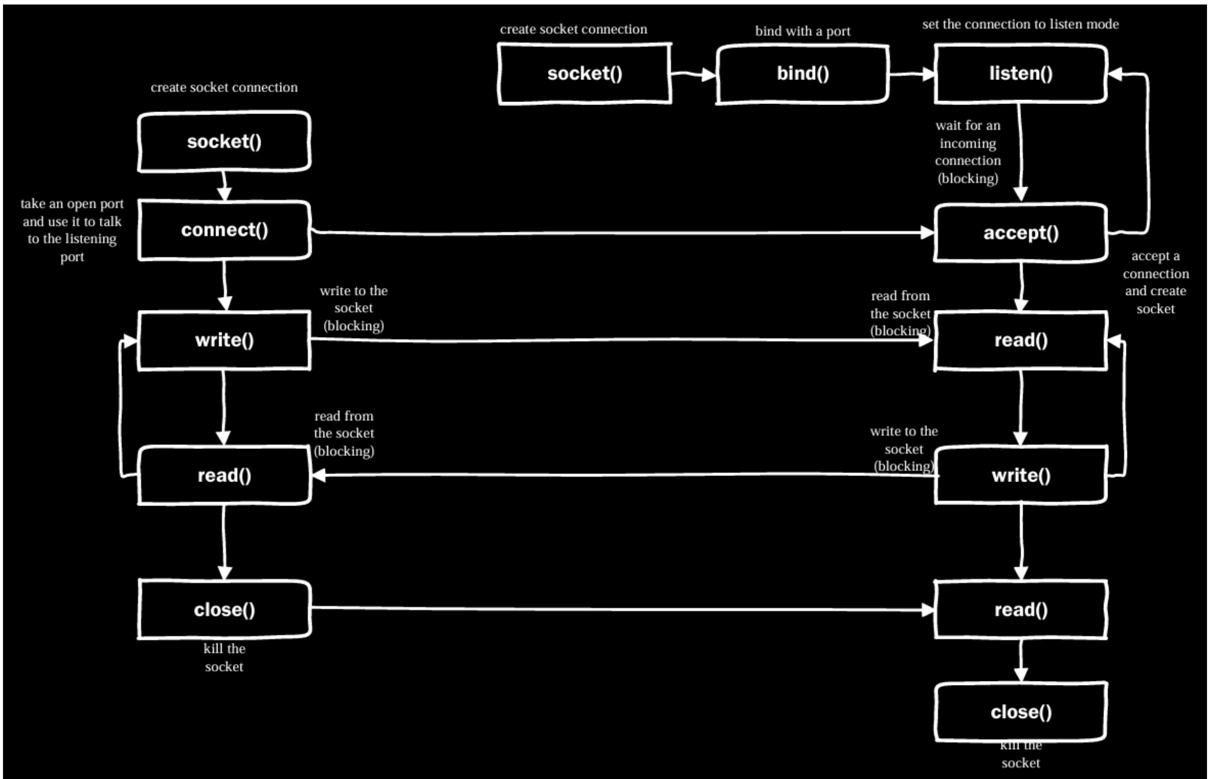
Raw Socket, Sequenced packet Socket etc....

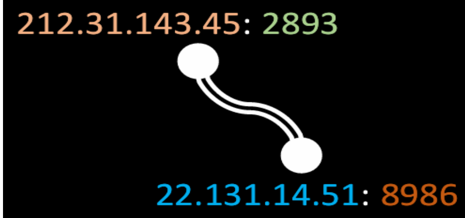
Socket

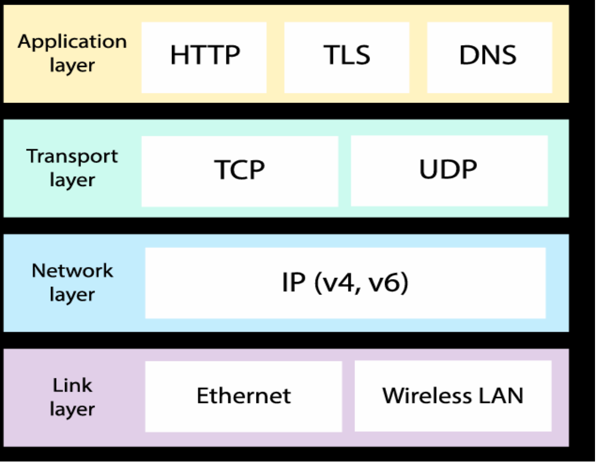
- How many types of socket?
- Characteristics of Socket
- Applications of sockets
- Other Sockets

Socket Programming

Socket programming is a way to enable communication between two systems (processes) over a network using **sockets**. It allows data exchange between a **client** and a **server**.


create socket connection
socket()
bind with a port
bind()
set the connection to listen mode
listen()
create socket connection
socket()
wait for an incoming connection (blocking)
take an open port and use it to talk to the listening port
connect()
accept()
accept a connection and create socket
write to the socket (blocking)
write()
read from the socket (blocking)
read()
read from the socket (blocking)
read()
write to the socket (blocking)
write()
close()
kill the socket
read()
close()
kill the socket





Application layer	HTTP	TLS	DNS
Transport layer	TCP	UDP	
Network layer	IP (v4, v6)		
Link layer	Ethernet	Wireless LAN	

REST API (Representational State Transfer API)
A **REST API** is a web-based API that follows REST (Representational State Transfer) principles, allowing communication between a client and a server using HTTP methods.

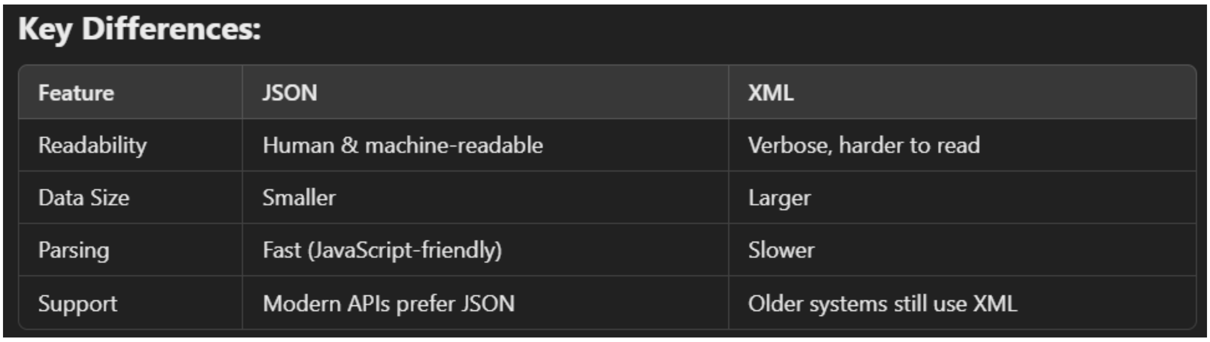
REST APIs typically return data in **two main response formats**:

1. JSON (JavaScript Object Notation) - Most Common

- Lightweight, easy to read.
- Used widely in web applications.
- Supports key-value pairs, arrays, and nested objects.

2. XML (Extensible Markup Language) - Less Common

- Older format, more structured.
- Used in legacy systems and some enterprise applications.
- Similar to HTML, with opening and closing tags.



Key Differences:

Feature	JSON	XML
Readability	Human & machine-readable	Verbose, harder to read
Data Size	Smaller	Larger
Parsing	Fast (JavaScript-friendly)	Slower
Support	Modern APIs prefer JSON	Older systems still use XML

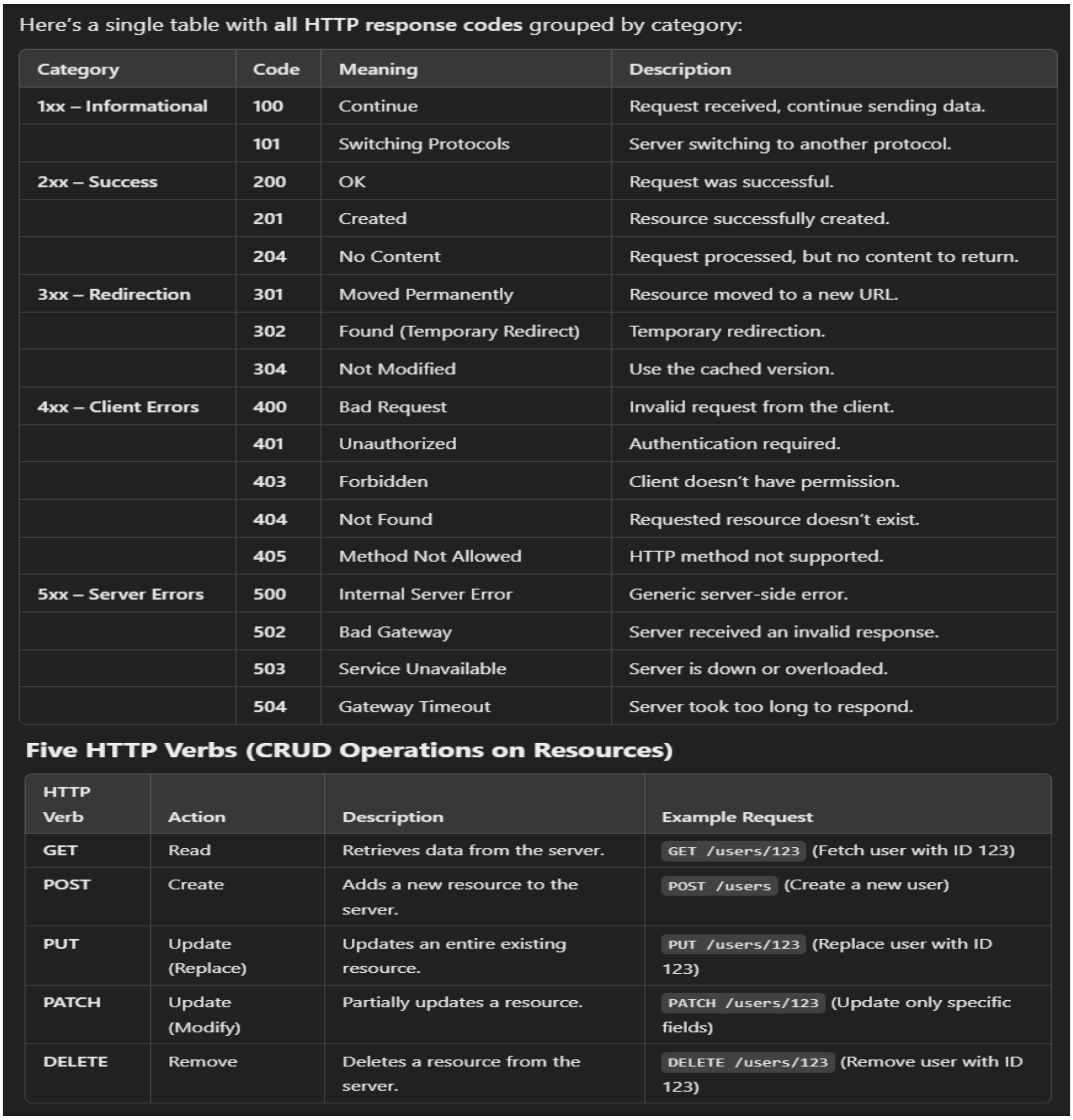
HTTP Request
An **HTTP request** is a message sent by a client (e.g., a web browser or mobile app) to a server to perform actions such as retrieving or modifying data.

1. HTTP Request Structure
An HTTP request consists of:

- **Request Line** – Method, URL, and HTTP version.
- **Headers** – Metadata (e.g., authentication, content type).
- **Body (Optional)** – Data sent in requests like POST or PUT.

2. HTTP Headers
Headers provide extra information in the request.
Common Headers:

- `Content-Type: application/json` → Specifies data format.
- `Authorization: Bearer <token>` → Authentication token.
- `User-Agent: Mozilla/5.0` → Identifies client type.



Here's a single table with **all HTTP response codes** grouped by category:

Category	Code	Meaning	Description
1xx – Informational	**100**	Continue	Request received, continue sending data.
	101	Switching Protocols	Server switching to another protocol.
2xx – Success	**200**	OK	Request was successful.
	201	Created	Resource successfully created.
	204	No Content	Request processed, but no content to return.
3xx – Redirection	**301**	Moved Permanently	Resource moved to a new URL.
	302	Found (Temporary Redirect)	Temporary redirection.
	304	Not Modified	Use the cached version.
4xx – Client Errors	**400**	Bad Request	Invalid request from the client.
	401	Unauthorized	Authentication required.
	403	Forbidden	Client doesn't have permission.
	404	Not Found	Requested resource doesn't exist.
	405	Method Not Allowed	HTTP method not supported.
5xx – Server Errors	**500**	Internal Server Error	Generic server-side error.
	502	Bad Gateway	Server received an invalid response.
	503	Service Unavailable	Server is down or overloaded.
	504	Gateway Timeout	Server took too long to respond.

Five HTTP Verbs (CRUD Operations on Resources)

HTTP Verb	Action	Description	Example Request
GET	Read	Retrieves data from the server.	`GET /users/123` (Fetch user with ID 123)
POST	Create	Adds a new resource to the server.	`POST /users` (Create a new user)
PUT	Update (Replace)	Updates an entire existing resource.	`PUT /users/123` (Replace user with ID 123)
PATCH	Update (Modify)	Partially updates a resource.	`PATCH /users/123` (Update only specific fields)
DELETE	Remove	Deletes a resource from the server.	`DELETE /users/123` (Remove user with ID 123)

Architecting a RESTful API

□API Specification

- Defines the API structure, including:
  - **Endpoints** (resource paths)
  - **Operations** (HTTP methods)
  - **Data Models** (request/response structures)
  - **Authentication** (security mechanisms)

2□Modeling the Resources

- Establishes the relationship between data models and API endpoints:
  - **Mapping fields to endpoints** (ensuring consistency)
  - **Defining input and output models** (structured request/response data)

3□Protecting Endpoints

- Implements security mechanisms to safeguard the API:
  - **Authentication protocols** (OAuth, JWT)

    - **Authorization & access scopes** (user roles, permissions)
    - **Privilege escalation prevention**
    - **SQL Injection protection**

4□Interactive Documentation

- Enhances API usability with:
    - **Automatic documentation generation** (e.g., Swagger)
    - **Testing tools** for API validation and integration

5□Caching

- Optimizes performance by reducing redundant requests:
    - **Idempotency** (ensuring safe retries)
    - **Database-level caching**
    - **Built-in cloud caching mechanisms**

6□Circuit Breaker

- Improves system resilience by handling failures effectively:
    - **Robustness** (ensuring API stability)
    - **Fail-safe retries** (automatically retrying failed requests)
    - **Fault tolerance mechanisms**
    - **Fail-fast approach** (stopping problematic requests early)

7□Debugging and Security Audit

- Ensures continuous monitoring and security assessment:
    - **Internal state tracking**
    - **Monitoring & alerting systems**
    - **Event logging for audits**
    - **Liveness and readiness checks**

---

here's a breakdown of the key points for architecting a RESTful API:

- **API Specification:**
  Define your API's structure using standards like the OpenAPI Specification (often implemented with Swagger). This step lays out all endpoints, operations, and data models, ensuring clarity and consistency from the start.
- **Modeling the Resources:**
  Develop clear data models that represent the resources your API will expose. Map these models to endpoints and ensure that both the input and output structures are consistent, making the API easier to understand and maintain.
- **Protecting End Points:**
  Secure your API by implementing robust authentication and authorization measures. This involves defining access scopes, using protocols to protect against common vulnerabilities (e.g., SQL injection), and ensuring that endpoints are only accessible to authorized users.
- **Interactive Documentation:**
  Use tools like Swagger to create interactive documentation. This allows developers to explore and test the API directly through the documentation, which improves developer experience and accelerates integration.
- **Caching:**
  Incorporate caching strategies—both at the database and cloud levels—to improve performance and reduce load. Effective caching helps ensure that the API can handle high traffic while maintaining fast response times.

- **Circuit Breaker:**
  Implement circuit breaker patterns to enhance system resilience. This design pattern helps manage failures gracefully, enabling the API to detect when a service is failing and to trigger safe fallback procedures or retries.
- **Debugging and Security Audit:**
  Ensure comprehensive logging, monitoring, and auditing are in place. This includes debugging tools, continuous monitoring of internal states, and regular security audits to detect and address vulnerabilities before they can impact the API’s operation.