# API OVERVIEW

* What is API?
* API Stands for **Application Programming Interface**
* API is like a messenger that helps two different applications talk to each other
* It defines a set of rules for sending requests and getting responses.
* API allows apps to share data and services without knowing each other’s internal details.
* For example, when a weather app shows live temperature, it gets that data from a **weather API**

* Types of APIs

**1. Open APIs (Public APIs)**

* These are **open for everyone** to use, sometimes free or with limited access.
* Developers use them to connect their apps with popular services.
* **Example:**
  + **Google Maps API** → lets any app show maps and locations.
  + **Weather API (OpenWeather)** → gives live weather data to apps.

**2. Internal APIs (Private APIs)**

* These are used **inside an organization only**, not shared with outsiders.
* Helps different teams or software in the same company connect smoothly.
* **Example:**
  + A company’s HR system uses an internal API to fetch employee details from the payroll system.
  + An e-commerce site’s backend uses internal APIs to connect its inventory, billing, and delivery systems.

**3. Partner APIs**

* Shared only with **trusted business partners** under an agreement.
* Safer than public APIs and used for collaboration between companies.
* **Example:**
  + A travel booking site (like MakeMyTrip) uses airline APIs (partner-only) to show real-time flight availability.
  + Payment gateways like Razorpay or PayPal share APIs with merchants to handle secure payments.

**4. Composite APIs**

* These **combine multiple APIs** into one single call.
* Saves time because instead of making many requests, you send only one.
* **Example:**
  + A food delivery app (like Zomato) uses a composite API to get restaurant details, menu, and delivery time in **one response**.
  + A banking app may use composite APIs to show **account balance + transaction history** together.
* **HTTP Status codes**
  + HTTP Stands for Hypertext Transfer Protocol
  + HTTP is the **protocol that allows data to travel over the web** between clients (browsers/apps) and servers.
  + When you open a website, your browser sends an **HTTP request** to the server, and the server sends back an **HTTP response**.
  + The status of the response is in form of code called HTTP Status codes
* HTTP status codes are **3-digit numbers** that the server sends back to the client to explain the result of the request.
* They tell whether your request worked, if there was an error, or if further action is needed.
* The codes are divided into **five main categories** based on the first digit:

1xx → Informational

* These codes indicate that the server has received the request and is continuing to process it.
* Mostly used in technical cases like protocols, rarely seen in normal API interactions.
* **Examples:**
  + **100 Continue:** Server has received the initial part of the request; client should continue.
  + **101 Switching Protocols:** Server is switching to a different protocol requested by the client.

🔹 2xx → Success

* This code indicates the request was successful, and the server returned the expected response.
* **Common Codes:**
  + **200 OK:** Request worked perfectly. Data is sent to the client.
    - Example: You requested a list of users → server sends all users.
  + **201 Created:** A new resource was successfully created on the server.
    - Example: You added a new book to the database via POST → server confirms creation.
  + **204 No Content:** Request succeeded but there’s no content to return.
    - Example: You deleted a file → server confirms deletion but sends nothing.

### 🔹 3xx → Redirection

* The requested resource has moved, and you may need to make another request to a different URL
* 3xx codes tell the client to go somewhere else to get the resource
* **Common Codes:**
  + **301 Moved Permanently:** The resource has a new permanent URL.
    - Example: Website permanently redirects from http://example.com → https://example.com.
  + **302 Found:** The resource is temporarily available at another URL.
    - Example: A sale page temporarily redirects users to a different URL for a promotion.

### 🔹 4xx → Client Errors

* There is a problem with the request from the client. The server cannot process it.
* 4xx codes mean the **client made a mistake**; correcting the request usually fixes it.
* **Common Codes:**
  + **400 Bad Request:** Request is incorrect or missing information.
    - Example: Sending a POST request without a required field, like a username.
  + **401 Unauthorized:** The client is not authenticated. Login or API key is required.
    - Example: Trying to access your Gmail inbox without logging in.
  + **403 Forbidden:** Client is authenticated but does not have permission.
    - Example: A regular user trying to access the admin panel.
  + **404 Not Found:** The requested resource does not exist.
    - Example: Requesting /user/999 when user 999 doesn’t exist.

### 🔹 5xx → Server Errors

* The client’s request was valid, but the **server failed** to process it.
* 5xx codes indicate **problems on the server side**, not the client side.
* **Common Codes:**
  + **500 Internal Server Error:** Server has a bug or crashed.
    - Example: A database failure caused the API to fail.
  + **502 Bad Gateway:** Server acting as a gateway got an invalid response from another server.
    - Example: API server fails while fetching data from another server.
  + **503 Service Unavailable:** Server is temporarily overloaded or down.
    - Example: Too many users visiting a website during a sale; server can’t handle the load.

**Response Formats**

**What a response format is:**

* The response format is the way a server packages data when it answers an API request.
* The client reads this format and converts it into objects it can use.
* Always check the Content-Type response header (for example application/json) to know how to parse it.

**JSON (JavaScript Object Notation)** is the most common format.

* It uses key–value pairs, is very lightweight, easy to read, and works well with almost all programming languages.
* For example, a user’s information in JSON looks like {"id": 1, "name": "Ravi", "email": "ravi@example.com"}.

**XML (Extensible Markup Language)** is an older format that uses tags to store data.

* It is more structured and supports attributes, but it is larger and more complex compared to JSON.
* A user’s information in XML looks like <user><id>1</id><name>Ravi</name><email>ravi@example.com</email></user>.

**HTML (HyperText Markup Language)** is mainly used to display web pages.

* Sometimes APIs return HTML when the response is meant to be shown directly on a browser.
* For example, a user’s information in HTML can look like <h1>Ravi</h1><p>Email: [ravi@example.com</p](mailto:ravi@example.com%3c/p)>.

**Plain Text** is the simplest form where the response is just raw text without any structure.

* It is usually used for small messages or status checks, such as OK or User: Ravi, Email: [ravi@example.com](mailto:ravi@example.com).

JSON is best for modern APIs, XML is useful for structured data but heavier, HTML is for web page responses, and Plain Text is for very simple outputs.

**Types of API Authentication Explained in Sentences**

**API Key** authentication works by giving a unique secret key to the client.

* This key is sent with every request, and the server uses it to check if the request is allowed.
* It is simple to use, but if the key is leaked, anyone can use the API.

**Basic Authentication** uses a username and password.

* The client sends these credentials (usually in an encoded form) with each request.
* It is easy to set up, but not very secure unless it is combined with HTTPS, because the credentials can be exposed.

**OAuth (Open Authorization)** is a token-based method that is more secure and widely used.

* Instead of sharing the actual password, the user gives permission, and the server issues a token that the client uses for access.
* Big platforms like Google and Facebook use OAuth so that apps can log in with your account safely.

**JWT (JSON Web Token)** is another token-based method, but the token itself contains the user’s data and permissions in a secure, encoded form.

* When a client makes a request, it sends the JWT, and the server verifies it without needing to look into a database each time.
* JWT is widely used in web and mobile applications because it is fast and secure.

**Versioning and Security Explained in Sentences**

**Versioning and Security in Simple Language**

* **Versioning:**  
  Versioning means giving numbers to your API when you update it.  
  This helps old apps to keep working even if you make changes.  
  For example, /v1/users is version 1 and /v2/users is version 2.  
  If you add new features or change the way the API works, you create a new version.  
  This way, developers can continue using the old version until they are ready to move to the new one.
* **Security:**  
  Security makes sure that data and APIs are safe from attackers.  
  Always use **HTTPS** so data is encrypted and cannot be stolen.  
  Use **authentication** methods like tokens or OAuth so only the right people can access the API.  
  Check user permissions (authorization) so everyone only does what they are allowed.  
  Validate the data that comes from users to stop hackers from sending harmful input.  
  Use **rate limiting** to stop misuse by too many requests at once.  
  Store keys and passwords safely and never share them in public.  
  Keep your system updated and fix security issues regularly.

**CRUD Operations**

CRUD means the four main actions we do with data: **Create, Read, Update, and Delete.**  
These actions are linked with HTTP methods in APIs.

**Create → POST**  
"Create" means adding new data to the server.  
When we use the **POST** method, we send new information that the server stores.  
Example: Adding a new student record in a database with their name and email.

**Read → GET**  
"Read" means fetching or viewing data from the server.  
The **GET** method is used to request information without changing anything.  
Example: Getting a list of all students or checking details of one student.

**Update → PUT/PATCH**  
"Update" means changing existing data on the server.  
The **PUT** method replaces the whole data with new data.  
The **PATCH** method updates only part of the data.  
Example: Updating a student’s email or correcting their name.

**Delete → DELETE**  
"Delete" means removing data from the server.  
The **DELETE** method is used to erase the data permanently.  
Example: Deleting a student record from the database.

**Optimization and Efficiency in APIs**

When you use an API, sometimes the response can be slow or the server can get overloaded, especially if there’s a lot of data. Optimizing APIs makes them faster, reliable, and less heavy. This is done by following processes:

**1. Pagination**

* When an API has a lot of data, sending everything at once is slow.
* Pagination splits data into smaller chunks (pages) so clients get it in parts.
* **Example:** Instead of sending 10,000 users at once, the API sends 100 users per page.

**2. Caching**

* Sometimes clients ask for the same data repeatedly.
* Caching **stores a copy** of the data temporarily so it can be sent faster next time.
* **Example:** First request fetches from the database, second request fetches from the cache, which is much faster.

**3. Minimize Data**

* Sending unnecessary information slows down the API and wastes bandwidth.
* Only include **what the client needs** in the response.
* **Example:** If the client only needs the user name and ID, don’t send full address, email, etc.

**4. Compression**

* Large responses take longer to send over the internet.
* Compression reduces the size of the data before sending, making it faster to transfer.
* **Example:** JSON data of 100 KB can be compressed to 20 KB using zip.

**5. Efficient Queries**

* The way the server fetches data affects speed.
* Writing **optimized queries** ensures the server gets data quickly without unnecessary processing.
* **Example:** Fetch only required fields from the database instead of selecting all columns.

**Requests Library in Python**

**What is it?**

* requests is a **popular Python library** used to communicate with APIs.
* It helps your Python program **send requests to servers** and get data back easily.
* You can use it to make all types of HTTP requests like **GET, POST, PUT, DELETE**.

**Uses:-**

* Without requests, making HTTP requests in Python is **complicated**.
* With requests, you can **write a few lines of code** to get data from any API.
* It also handles **responses, status codes, headers, and JSON parsing** automatically.

**Basic Example**

import requests

# Send a GET request to the API

response = requests.get("https://api.example.com/users/1")

# Check the status code of the response

print(response.status\_code) # 200 means request was successful

# Get the response data in JSON format

print(response.json()) # Example output: {'name': 'Purva', 'age': 20}

**Explanation:**

1. requests.get() → Sends a request to the given URL.
2. response.status\_code → Tells whether the request was successful (200) or failed.
3. response.json() → Converts the API response into a Python dictionary for easy use.

**RBAC (Role-Based Access Control)**

**What is RBAC?**

* RBAC is a system used to **control who can access what** in an application or API.
* Instead of giving everyone full access, **permissions are assigned based on roles**.
* Each role has a **set of allowed actions**, like read, write, or delete.

**Use of RBAC**

* Not every user should access all data or perform all actions.
* Protects **sensitive data** from unauthorized users.
* Makes managing permissions easier because you assign access to roles, not individual users.

**How It Works (Example)**

1. First we have to define roles like Admin, User, Manager, Guest.
2. **Permissions:** Each rolesis assigned by its different work. We have to assign what each role can do.
   * **Admin:** Can read, write, update, delete everything
   * **User:** Can only read data
   * **Guest:** Can read only public data
3. **Users:** Assign a role to each user.
   * Example: Alice → Admin, Bob → User