5. REST APIs

Introduction To APIs

- **API:** Set of **rules** that define how applications or devices can **communicate** with each other
- A web API is an **entry-point** between **clients and resources** on the web
  - **Clients**
    - Users who want to access information from the web
    - The client can be a person or a software system that uses the API
  - **Resources**
    - Resources can be images, videos, text, numbers, or any type of data
    - The machine that gives the resource to the client is also called the server
- Restaurant Analogy
  - Imagine you are sitting in a restaurant and have selected your order from the menu. Your waiter communicates your order to the kitchen and returns the food back to you.
    - Waiter → API (As it communicates a request from one device to another)
    - Menu → API documentation (Each API has documentation that outlines the requests you are allowed to make, and the type of response you should expect to recieve)

REST (**RE**presentational **S**tate **T**ransfer)

- REST
  - An **architectural style** that defines how an API should **work**
- REST vs SOAP (**S**imple **O**bject **A**ccess **P**rotocol)
  - SOAP or XML-RPC: Imposes a **strict framework** on developers
  - REST APIs: Can be developed using **any programming language**. Supports a variety of data formats. It is **simple, flexible** and uses **less bandwidth** making it more suitable for internet usage
- REST APIs
  - APIs that follow the REST architectural style
  - All communication done via REST API uses only **HTTP request**

REST Design Principles

1. Uniform Interface

- All **requests** for the **same** resource should look the same, no matter where the request comes from
- The server transfers information in a standard format which can be **different** from the internal representation of the resource on server.
    - For example, the server can store data as **text** but send it in an **HTML** representation format
- Uniform interface imposes four architectural constraints:
    1. Requests should use a **uniform resource identifier (URI)** to specify resources
    2. Clients can modify or delete resources as needed, as the server provides metadata in the resource representation that describes the resource sufficiently
    3. Servers send messages with metadata that guide clients on how to **process** the resource representation effectively
    4. Servers include **hyperlinks** in the resource representation, allowing clients to dynamically discover and access any additional resources needed to complete a task

2. Statelessness

- Server applications **aren’t** allowed to store any data related to a **client request**
  - Server completes every client request **independently** of all **previous requests**
  - Therefore, each request needs to include all the information necessary for processing it

3. Layered system

- There may be a number of different intermediaries in the communication loop between a client and a server
- We can design RESTful web service to run on several servers with multiple layers such as security, application, and business logic, working together to fulfill client requests. These layers remain invisible to the client

4. Cacheability

- Process of storing some responses on the client or on an intermediary to **improve** server **response** time
  - For example, suppose that you visit a website that has common header and footer images on every page. Every time you visit a new website page, the server must resend the same images. To avoid this, the client caches or stores these images after the first response and then uses the images directly from the cache.
- RESTful services manage caching by marking API responses as cacheable or non-cacheable, allowing clients to know which data can be stored for reuse

5. Code on demand

- Servers can enhance or adjust client functionality by sending executable code to the client temporarily
  - **Example**: When filling out a registration form on a website, the browser highlights errors (e.g., incorrect phone numbers) instantly. This real-time validation happens due to the code the server sends, enabling immediate client-side error handling.

Benefits of REST API?

- **Optimized Client-Server Interaction**
  - Statelessness removes server load as it does not have to retain past client request information
  - Well-managed caching partially or completely eliminates some client-server interactions
- **Flexibility**
  - By supporting total client-server separation
    - Technology changes at the server application do not affect the client application
  - By allowing layered architecture
    - For example, developers can make changes to the database layer without rewriting the application logic

How Do REST API Work?

- Similar as **browsing the internet**
    - The client contacts the server by using the API when it requires a resource
- These are the general steps for any REST API call:
    - The client sends a request to the server.
        - Client follows the API documentation to format the request in a way that the server understands
    - The server authenticates the client and confirms that the client has the right to make that request
    - The server processes the request internally
    - The server returns a response to the client
        - The response also includes information that the client requested along with the information whether the request was successful

What Does the REST API Client Request Contain?

RESTful APIs require requests to contain the following main components:

1. **Unique Resource Identifier**
   - Server identifies resources using a **Uniform Resource Locator (URL)**
     - The URL specifies the path to the resource
     - A URL is similar to the website address that you enter into your browser to visit any webpage
     - A URL is also called the request endpoint
2. **Method**
   - An HTTP method tells the server what it needs to do to the resource
   1. **GET**
      - To **access resources** that are located at the specified URL on the server
      - Client can cache GET requests and send parameters in the REST API request to instruct the server to filter data before sending

2. **POST**
    - To **send data** to the server
    - Client include the data representation with the request
    - Sending the same POST request multiple times has the side effect of creating the same resource multiple times
3. **PUT**
    - To **update existing resources** on the server
    - Unlike POST, sending the same PUT request multiple times gives the same result
4. **DELETE**
    - To **remove** a resource on the server
    - A DELETE request can change the server state. However, if the user does not have appropriate authentication, the request fails.

3. HTTP Headers

- **Metadata e**xchanged between the client and server
  1. **Data**
     - Client requests might include data for the POST, PUT, and other HTTP methods to work successfully
  2. **Parameters**
     - Parameters to give the server more details about what needs to be done
     - The following are some different types of parameters:
       - **Path** parameters that specify URL details
       - **Query** parameters that request more information about the resource
       - **Cookie** parameters that authenticate clients quickly

What does the REST API Server response contain?

1. **Status Line**
    - The status line contains a **three-digit status code** that communicates request success or failure
        - **2XX** → success
        - **3XX** → URL redirection
        - **4XX** and **5XX** → errors
    - The following are some common status codes:
        - 200: Generic success response
        - 201: POST method success response
        - 400: Incorrect request that the server cannot process
        - 404: Resource not found

2. Mescsage Body

- ontains **the resource representation**
- Server selects an appropriate representation **format** based on what the request headers contain
- Clients can request information in **XML** or **JSON** formats, which define how the data is written in plain text. For example, if the client requests the name and age of a person, the server returns a JSON representation as follows:
  - '{"name":"John", "age":30}'

3. Headers

- **Metadata** about the response
- They give more context about the response and include information such as the server, encoding, date, and content type

What are REST API Authentication methods?

- A RESTful web service must **authenticate** (verify identity) **requests** before it can send a response
- RESTful API has three common authentication methods:

1. **HTTP Authentication**
    - Basic Authentication
        - The client **sends the user name and password** in the request header
        - It encodes them with **base64**, which is an encoding technique that converts the pair into a set of 64 characters for safe transmission.
    - Bearer Authentication
        - The process of **giving access control** to the token bearer
        - The bearer token is typically an encrypted string of characters that the server generates in response to a login request. The client sends the token in the request headers to access resources

2. **API Keys**
   - The server assigns a unique generated value to a first-time client
   - Whenever the client tries to access resources, it uses the unique API key to verify itself
   - API keys are less secure because the client has to transmit the key, which makes it vulnerable to network theft
3. **OAuth**
   - OAuth combines passwords and tokens for highly secure login access to any system
   - The server first requests a password and then asks for an additional token to complete the authorization process. It can check the token at any time and also over time with a specific scope and longevity

JavaScript APIs

- Most JavaScript APIs follow the **REST** architectural style
- These are referred to as RESTful APIs, and follow the CRUD paradigm:
  - Four basic functionalities needed when communicating between services and with a database
    - **C**reate
    - **R**ead
    - **U**pdate
    - **D**elete
  - These CRUD operations are often aliased as follows:
    - Create → POST
    - Read → GET
    - Update → PUT
    - Delete → DELETE

Example: Imagine an API which communicates with a **banking service** to process **online payments**

Method	URL	Description
POST	api/customer	● **Create** a new banking customer ● Depending on the specifications of the API, this may include options to provide data for this customer, such as a name or credit card details. ● It may also automatically generate new information upon creation, such as an id.
GET	api/customers/{id}	● **Retrieve** the information of a customer ● The API assumes a unique id for each customer, that is used in the URL to specify which customer's information you are searching. ● A response for this request can come in many different formats, such as JSON or XML, depending on the API

PUT	api/customers/{id}	● **Update** information for a specific customer
		● This will overwrite the current data with new data
		● Some APIs allow you to include a "body" in which you can specify a load of data to be sent with the request
		● Example: `Update a customer's first and last name`
		{
		"first_name": "Thomas",
		"last_name": "Watson"
		}

DELETE	api/customers/{id}	● **Delete** a banking customer

Examples of JavaScript APIs

- **Document Object Model (DOM) API**
  - One of the most basic JavaScript APIs
  - It connects web pages to scripts by representing the **structure of a webpage** in memory making it accessible for modfication as required
- **XMLHttpRequest (XHR)**
  - Allows you to retrieve data without refreshing the entire page
  - This is important when you to want to update only a part of a page without disrupting what a user is currently doing on the page

Summary