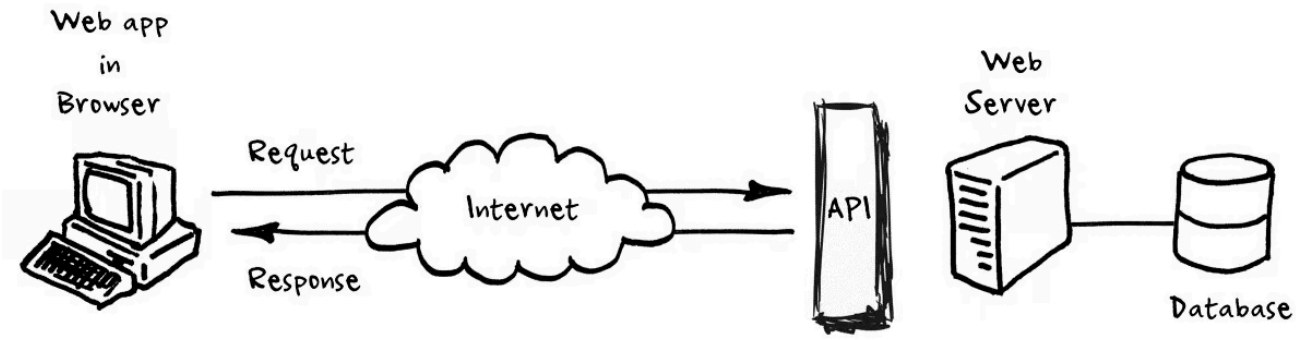

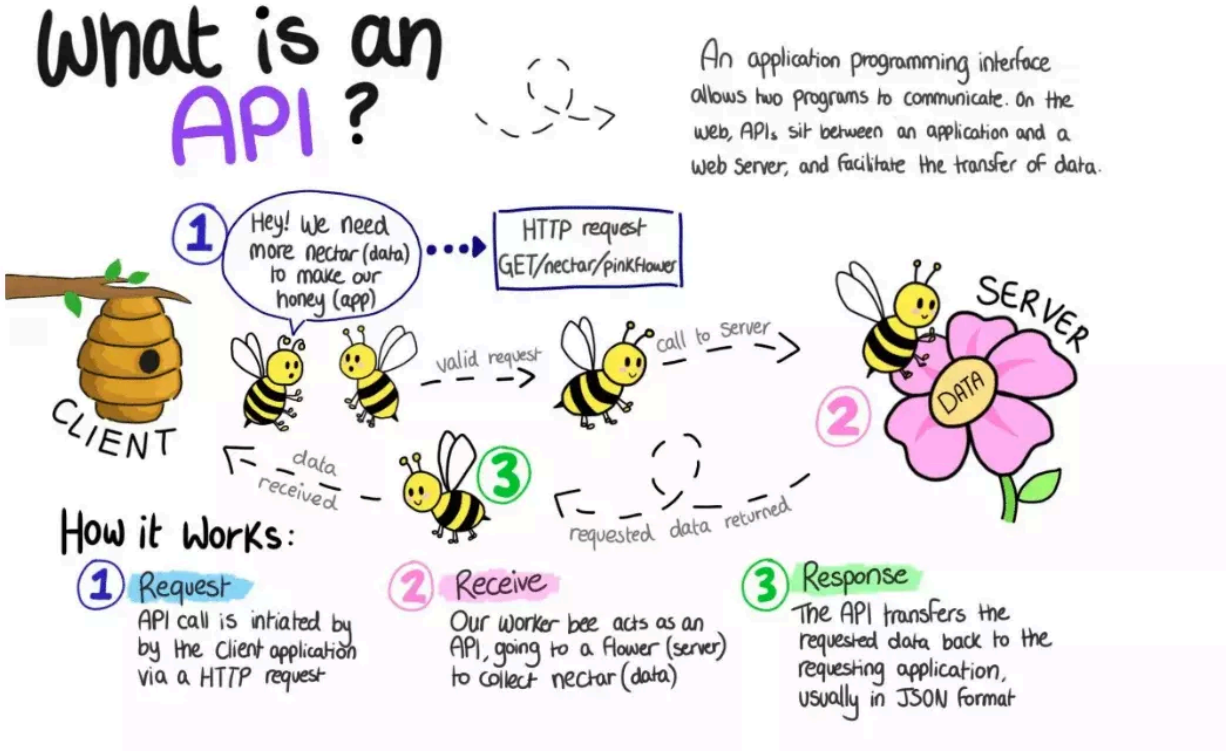

API Programming: Comprehensive Lecture Notes

API Fundamentals and Request Handling

What is an API?

An Application Programming Interface (API) is a set of rules and protocols that allows different software applications to communicate with each other. APIs define the methods and data formats that applications can use to request and exchange information.

Think of an API as a waiter in a restaurant:

- You (the client) don't go directly into the kitchen (the server's internal systems)
- Instead, you give your order to the waiter (the API)
- The waiter takes your request to the kitchen and returns with your food (the data)

Core API Concepts

APIs are fundamental building blocks in modern software development for several reasons:

1. **Abstraction**: APIs hide complex implementation details while exposing only necessary functionality
2. **Reusability**: Well-designed APIs can be used across multiple applications
3. **Modularity**: Systems can be built as collections of independent services connected via APIs
4. **Scalability**: API-based architectures can scale horizontally by adding more instances of services
5. **Security**: APIs provide controlled access to resources, implementing authentication and authorization

Types of APIs

1. **Web APIs**: Accessible over HTTP/HTTPS (REST, GraphQL, SOAP)
2. **Library/Framework APIs**: Programming interfaces provided by code libraries
3. **Operating System APIs**: Interfaces to interact with OS functionalities
4. **Database APIs**: Interfaces to interact with databases (JDBC, ODBC)
5. **Hardware APIs**: Interfaces to interact with hardware components

REST API Architecture

REST (Representational State Transfer) is an architectural style for designing networked applications. RESTful APIs have become the standard for web services due to their simplicity and scalability.

Key Principles of REST

1. **Stateless Communication**: Each request from client to server must contain all information needed to understand and process the request
2. **Client-Server Separation**: Client and server evolve independently
3. **Uniform Interface**: Standardized ways to interact with resources through:
   - Resource identification (URLs)
   - Resource manipulation through representations
   - Self-descriptive messages
   - Hypermedia as the engine of application state (HATEOAS)
4. **Layered System**: Intermediary servers can be used for load balancing, caching, etc.
5. **Cacheable**: Responses must define themselves as cacheable or non-cacheable

Resources and URLs

In REST, everything is considered a resource, which can be:

- A document (e.g., an article)
- A service (e.g., "today's weather in New York")
- A collection of resources (e.g., all users)

Resources are identified by URLs (Uniform Resource Locators), following conventions like:

```
https://api.example.com/v1/resources/collection/item
```

Best practices for REST URLs:

- Use nouns, not verbs (e.g., `/users` not `/getUsers`)
- Use plural nouns for collections (e.g., `/users` not `/user`)
- Use hierarchical relationships (e.g., `/users/123/orders`)
- Include API versioning (e.g., `/v1/users`)

HTTP Methods

HTTP defines several methods (also called "verbs") that indicate the desired action to be performed on a resource:

Core HTTP Methods

1. **GET**: Retrieve data from a specified resource
   - Safe: Doesn't change server state
   - Idempotent: Multiple identical requests have same effect as a single request
   - Example URL: `GET /api/users/123`
2. **POST**: Submit data to create a new resource
   - Not safe: Changes server state
   - Not idempotent: Multiple identical requests create multiple resources
   - Example URL: `POST /api/users` with user data in request body
3. **PUT**: Update an existing resource (or create if it doesn't exist)
   - Not safe: Changes server state
   - Idempotent: Multiple identical requests have same effect as a single request
   - Example URL: `PUT /api/users/123` with updated user data in request body
4. **DELETE**: Remove a specified resource
   - Not safe: Changes server state
   - Idempotent: Multiple identical requests have same effect as a single request
   - Example URL: `DELETE /api/users/123`

Additional HTTP Methods

5. **PATCH**: Partially update a resource
   - Example URL: `PATCH /api/users/123` with partial user data
6. **HEAD**: Same as GET but returns only HTTP headers, no body
   - Useful for checking if a resource exists or has been modified
7. **OPTIONS**: Returns supported HTTP methods for a specified URL
   - Useful for CORS preflight requests

API Request/Response Cycle

Understanding the complete flow of an API request is essential for effective API development and troubleshooting.

Request Components

1. **HTTP Method**: Defines the action (GET, POST, etc.)
2. **URL/Endpoint**: Identifies the resource
3. **Headers**: Metadata about the request (content type, authentication, etc.)
4. **Query Parameters**: Additional data passed in the URL (e.g., `?key=value`)
5. **Request Body**: Data sent to the server (typically JSON for modern APIs)

Response Components

1. **Status Code**: Numeric code indicating success/failure (200, 404, 500, etc.)
2. **Headers**: Metadata about the response (content type, caching directives, etc.)
3. **Response Body**: Data returned by the server (typically JSON for modern APIs)

Example Request/Response Cycle

1. Client prepares a request to create a new user:

```
POST /api/users
Content-Type: application/json
Authorization: Bearer eyJhbGc...

{
  "name": "John Doe",
  "email": "john@example.com"
}
```

2. Request is sent over the network to the server
3. Server receives and processes the request:
   - Validates authentication token
   - Parses JSON body
   - Validates input data
   - Creates user in database
   - Prepares response
4. Server sends response back to client:

```
201 Created
Content-Type: application/json
Location: /api/users/456

{
  "id": 456,
  "name": "John Doe",
  "email": "john@example.com",
  "created_at": "2025-04-15T10:30:00Z"
}
```

5. Client processes the response:
   - Checks status code (201 means resource created)
   - Parses JSON response
   - Updates UI or takes next action

HTTP Status Codes

Status codes are an important part of the API response. They provide immediate feedback on the result of a request.

Status Code Categories

1. **1xx (Informational)**: Request received, continuing process
   - 100: Continue
   - 101: Switching Protocols

2. **2xx (Success)**: Request successfully received, understood, and accepted
   - 200: OK - Standard success response
   - 201: Created - Resource created successfully
   - 202: Accepted - Request accepted but processing not complete
   - 204: No Content - Success but nothing to return (often used for DELETE)
3. **3xx (Redirection)**: Further action needed to complete request
   - 301: Moved Permanently
   - 302: Found (Temporary Redirect)
   - 304: Not Modified (used with conditional GET requests)
4. **4xx (Client Error)**: Error caused by the client
   - 400: Bad Request - Generic client error
   - 401: Unauthorized - Authentication required
   - 403: Forbidden - Authentication succeeded but user lacks permission
   - 404: Not Found - Resource doesn't exist
   - 405: Method Not Allowed - HTTP method not supported for this resource
   - 409: Conflict - Request conflicts with current state of the resource
   - 422: Unprocessable Entity - Request understood but semantically incorrect
5. **5xx (Server Error)**: Error on the server side
   - 500: Internal Server Error - Generic server error
   - 502: Bad Gateway - Server acting as gateway received invalid response
   - 503: Service Unavailable - Server temporarily unable to handle request
   - 504: Gateway Timeout - Gateway timeout waiting for response

---

📜 Understanding HTTP Headers

🌐 What Are HTTP Headers?

HTTP headers are **key-value pairs** sent along with every HTTP request and response. They provide metadata about the request/response — such as content type, authentication, caching rules, user agent, etc.

They **don't contain the main content**, but rather **instructions or context** about the content.

Sure! Here's a detailed **Markdown (MD) note** titled **"Understanding HTTP Headers"** — designed for clear learning and quick referencing.

📜 Understanding HTTP Headers

🌐 What Are HTTP Headers?

HTTP headers are **key-value pairs** sent along with every HTTP request and response. They provide metadata about the request/response — such as content type, authentication, caching rules, user agent, etc.

They **don't contain the main content**, but rather **instructions or context** about the content.

🔥 Why Are Headers Important?

1. ✅ **Authentication & Security**
   Example: Pass tokens, cookies, or API keys to authenticate users.

```
Authorization: Bearer <token>
```

2. 📦 **Content Description & Negotiation**
   Tell the server what format you expect (like JSON, XML, HTML).

```
Accept: application/json
Content-Type: application/json
```

3. 🚀 **Custom Information**
   Some APIs use custom headers like:

```
X-User-ID: 123
X-Client-Version: 1.0.5
```

4. 📶 **Caching & Control**
   Control how responses are cached or revalidated.

```
Cache-Control: no-cache
If-None-Match: "abc123"
```

📦 Commonly Used Headers

Header	Purpose
`Authorization`	Sends credentials (token, basic auth)
`Content-Type`	Declares format of request body
`Accept`	Informs server what response format is expected
`User-Agent`	Identifies the client software
`Cache-Control`	Tells cache systems how to cache
`X-*`	Custom headers used by APIs

🤔 How to Know What Headers You Need?

- 📘 **Read the API Documentation** - The API usually tells you required headers.
- 🕵️ **Use Browser Dev Tools** - Open Dev Tools → Network tab → Inspect headers.
- 🔧 **API Tools like Postman or Insomnia** - They suggest required headers or add them automatically.
- 🍼 **Trial and Error** - Some APIs return helpful errors when headers are missing.

Example in Python (`httpx`)

```python
import httpx

headers = {
    "Authorization": "Bearer my_api_token",
    "Accept": "application/json"
}

response = httpx.get("https://api.example.com/data", headers=headers)

print(response.status_code)
print(response.json())
```

Bonus: Setting Headers with `requests`

```python
import requests

headers = {
    "User-Agent": "MyApp/1.0",
    "Accept": "application/json"
}

response = requests.get("https://api.example.com/info", headers=headers)
```

What Happens Without Headers?

- You may get a `401 Unauthorized` or `403 Forbidden` error.
- The server may respond in a different format than expected.
- Requests may be **rejected**, **delayed**, or **mishandled**.

Summary

- Headers are metadata for HTTP transactions.
- They are essential for **security**, **data format**, and **custom behavior**.
- Learn to read API docs and debug headers for success in web/API work.