REST Services and Express JS

Eishan Weerasinghe

Learning Outcomes

- ❑ Describe the main components of three tier architecture for web development.
- ❑ Understand the concept of RESTful web services and how they work
- ❑ Understand the HTTP methods used in RESTful services (GET, POST, PUT, DELETE)
- ❑ Understand the different HTTP status codes and their meanings
- ❑ Know how to interpret HTTP responses using status codes
- ❑ Understand the basics of Express.js framework
- ❑ Be able to create and configure an Express.js server
- ❑ Understand how to use middleware in Express.js to enhance their API's functionality.

Web Sever and Web Service

Aspect	Web Server	Web Service
Definition	A software application that handles client requests and serves web content.	A software system that enables applications to communicate over a network.
Primary Function	Delivers web content like HTML pages, images, scripts, and other resources.	Exposes data and functionality for other applications to consume.
Communication Protocol	Uses HTTP/HTTPS to serve content to clients (e.g., browsers).	Uses standardized protocols like REST (HTTP-based) or SOAP (XML-based).
Content Delivery	Sends static or dynamic content directly to users (e.g., webpages).	Provides data or services (e.g., APIs) to applications, not end-users directly.
Examples	Apache, Nginx, Microsoft IIS.	RESTful APIs (e.g., Twitter API), SOAP-based services (e.g., payment gateways).
Accessibility	Accessed by web browsers or devices requesting web resources.	Consumed by web browsers, mobile apps, or other web services/applications.
Data Format	Primarily delivers HTML, CSS, JavaScript, images, etc.	Typically delivers data in formats like JSON, XML, or plain text.
Use Case	Hosting websites, serving files, managing web traffic.	Enabling interoperability between systems (e.g., app-to-app communication).
Interaction	Client-server model (browser requests, server responds).	Application-to-application model (e.g., one service calls another).

Web Server

- A **software application** that handles client requests and serves web content.
- Delivers **HTML pages**, **images**, **scripts**, **and other resources** over the internet or an intranet.
- Communicates using the **HTTP/HTTPS protocol**.
- Examples: **Apache**, **Nginx**, **Microsoft IIS**.

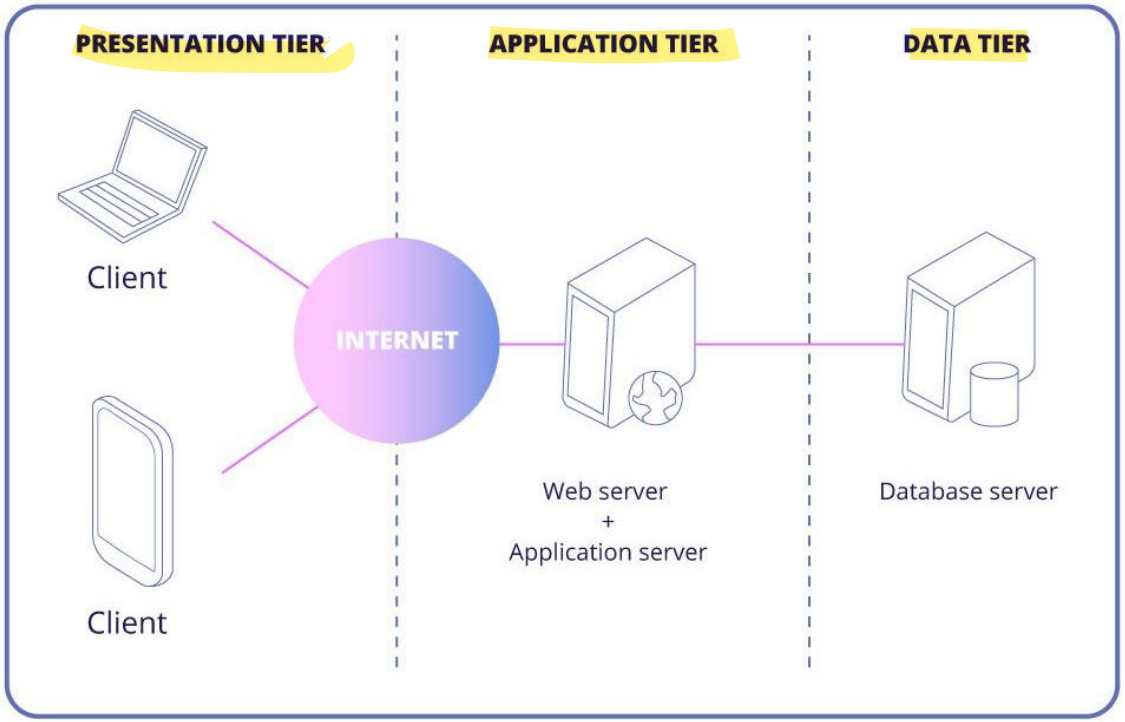
Web Service

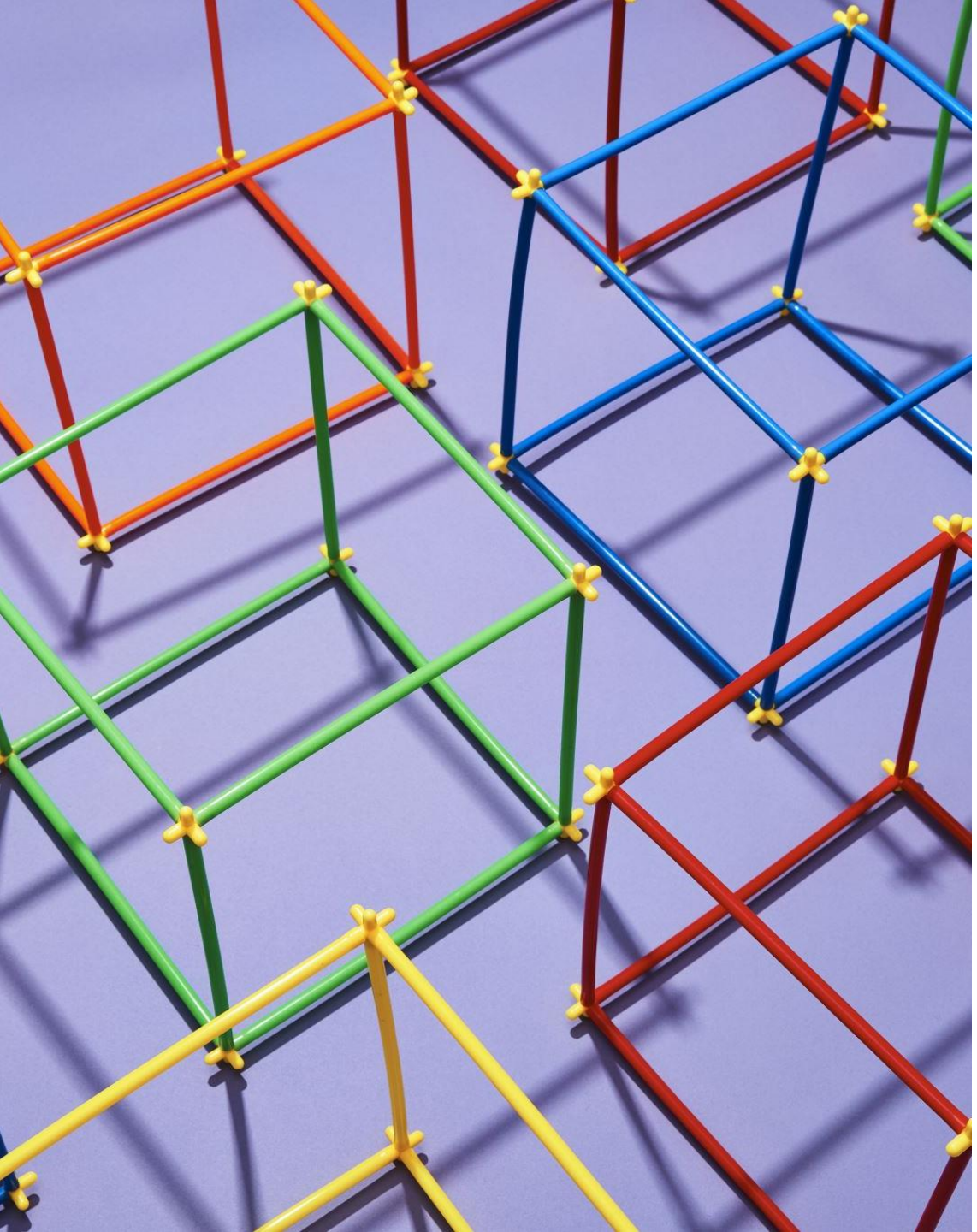
- A **software system** that enables applications to communicate over a network.
- Uses **standardized protocols** like **REST (HTTP-based)** **and** **SOAP (XML-based)**.
- Exposes **data and functionality** that can be accessed by different platforms and programming languages.
- Can be consumed by **web browsers**, **mobile apps**, **or other web services**.

Web Sever and Web Service

Feature	***Web Server***	***Web Service***
Purpose	Hosts websites and serves files	Exposes functionality or data to other applications
Communication	Handles HTTP requests/responses	Uses APIs like REST, SOAP, or GraphQL
Data Format	Serves HTML, CSS, JavaScript	Sends/receives data in JSON, XML
Example	Apache, Nginx, IIS	REST API, SOAP API

Three Tier Architecture





Three Tier Architecture

- Modern web applications are engineered based on this architecture.
- Has three tiers.
  - **Presentation tier/ layer**
  - **Application tier/ layer**
  - **Database tier/ layer**
- The most popular form of n-tier (multitier) architecture.

Three Tier Architecture

Why Choose Three-Tier Architecture for Web Applications?

- ***Key Benefits:***

✅ **Independent Infrastructure** – Each layer (Presentation, Application, and Data) runs on its own infrastructure, ensuring better performance and maintainability.

✅ **Parallel Development** – Different teams can develop each tier **independently**, improving efficiency and speeding up the development process.

✅ **Scalability** – You can **scale up or down** each layer separately, without affecting the other tiers, making resource management more flexible.

✅ **Enhanced Security** – Unlike two-tier architecture, this model provides **better security** by **isolating data and logic from the frontend**, reducing vulnerabilities.

Presentation Layer

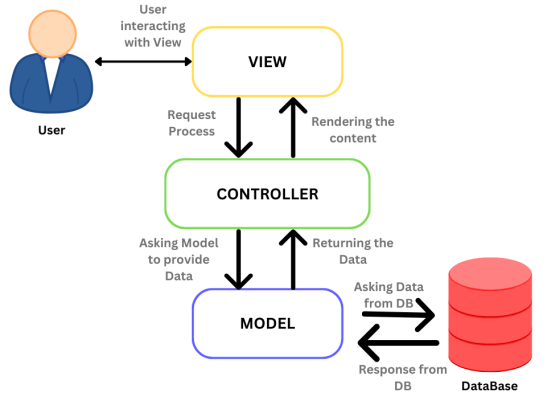
client side / front side

- The **Presentation Layer**, also called the **client-side** or **frontend**, is responsible for interacting with users.
- It is the **visible part** of the application where users input data and receive outputs.
- The **main goal** of this layer is to ensure a **smooth user experience (UX)** with responsive and accessible interfaces.
- **Modern Frontend Frameworks and Libraries:**

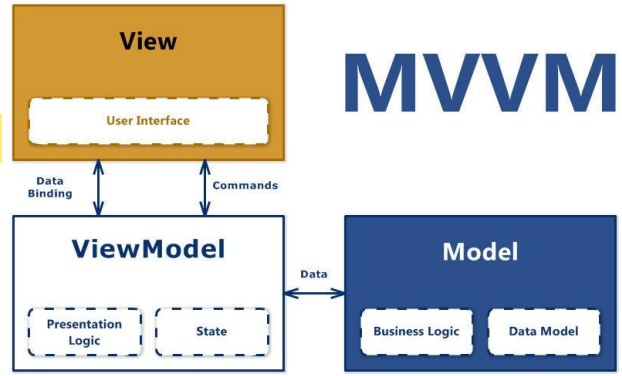
  React.js, Angular, Vue.js.
- **Legacy Frontend Technologies:**

  jQuery, AJAX (Asynchronous JavaScript and XML).

Architecting the Presentation Layer

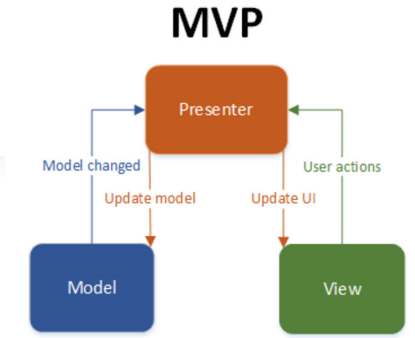


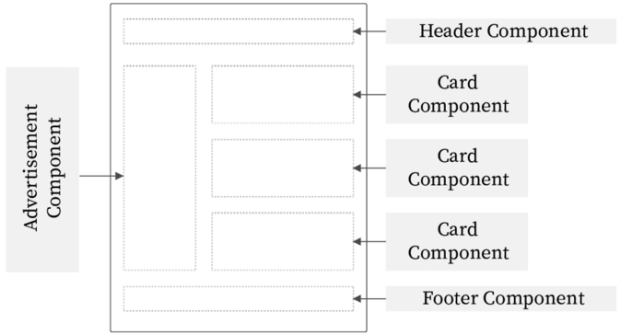
- **Model-View-Controller (MVC)**
  - Separates the application into **Model (data), View (UI), and Controller (logic)**.
  - Enhances **code reusability and maintainability**.
  - Used in frameworks like **Spring MVC (Java), ASP.NET MVC (.NET), Django (Python)**.
- **Model-View-ViewModel (MVVM)**
  - A variation of MVC where the **ViewModel** acts as a mediator between Model and View.
  - Reduces UI dependencies on business logic.
  - Common in **Angular, Vue.js, and WPF (Windows Presentation Foundation)**.

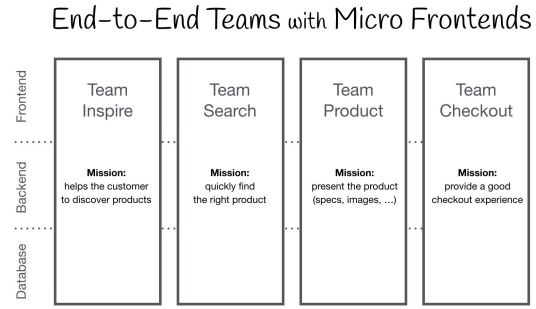


Architecting the Presentation Layer

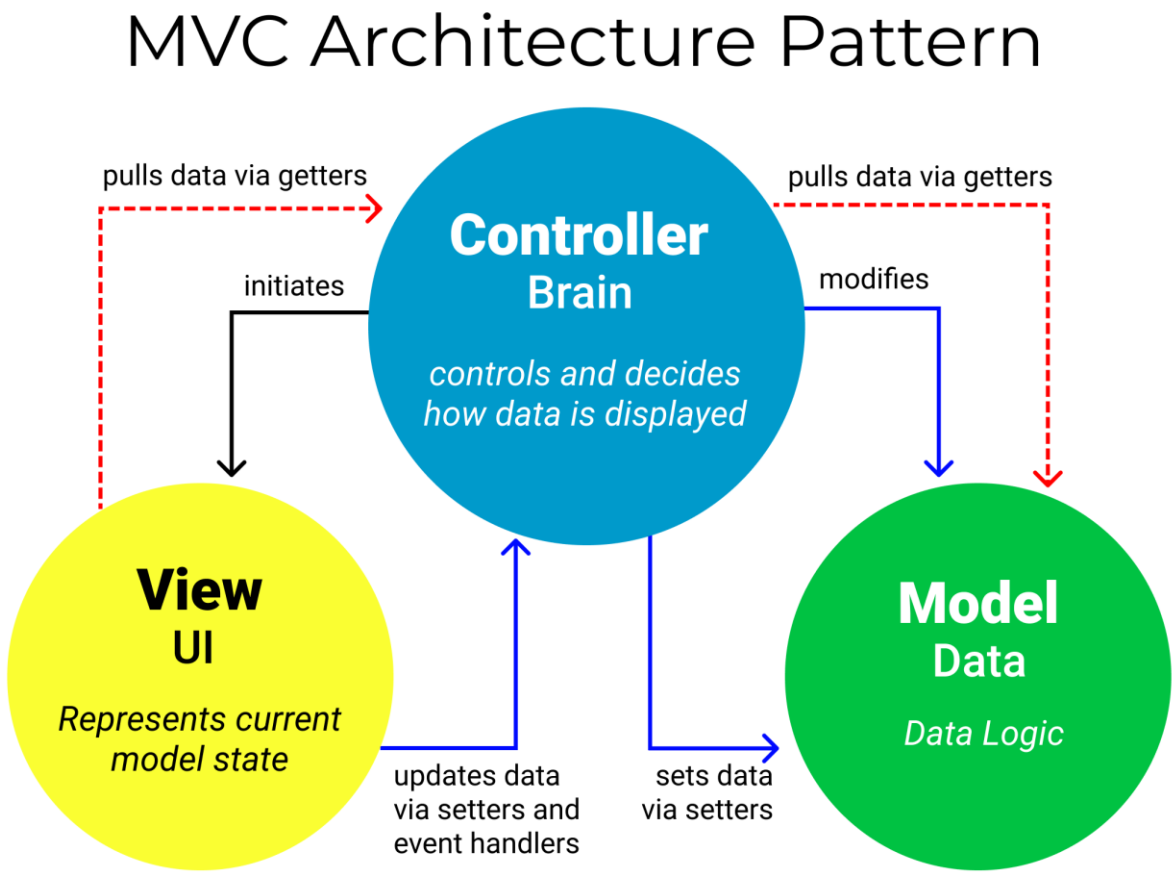
- **Model-View-Presenter (MVP)**
  - The **Presenter** handles business logic instead of the Controller.
  - Used in **desktop and mobile application development**.
- **Component-Based Architecture**
  - Modern frontend frameworks break the UI into **reusable components**.
  - Enhances modularity and scalability.
  - Used in **React.js, Angular, Vue.js**.
- **Micro Frontends**
  - Divides the frontend into **small, independent applications**.
  - Allows multiple teams to work on different parts of the UI.
  - Ideal for **large-scale, distributed applications**.



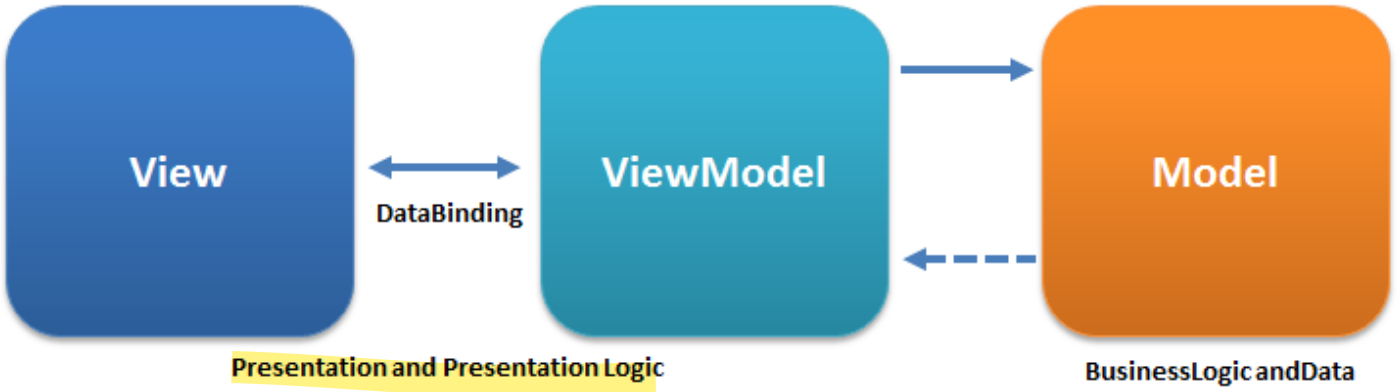




Architecting the Presentation Layer: MVC



Architecting the Presentation Layer: MVVM



server side / back end

Application Layer

- The Application Layer, also called the **server-side** or **backend**, is responsible for processing **business logic, handling requests, and managing data**.
- It acts as an **intermediary** between the **Presentation Layer (frontend)** and the **Database Layer.**
- Technologies used to implement include but not limited to,

💻 **Programming Languages & Frameworks:**

- **Java** – Spring, Spring Boot
- **Node.js** – Express.js, Koa
- **Python** – Django, Flask
- **PHP** – Laravel, Symfony
- **.NET** – ASP.NET Core

🛠 **Infrastructure & Deployment**

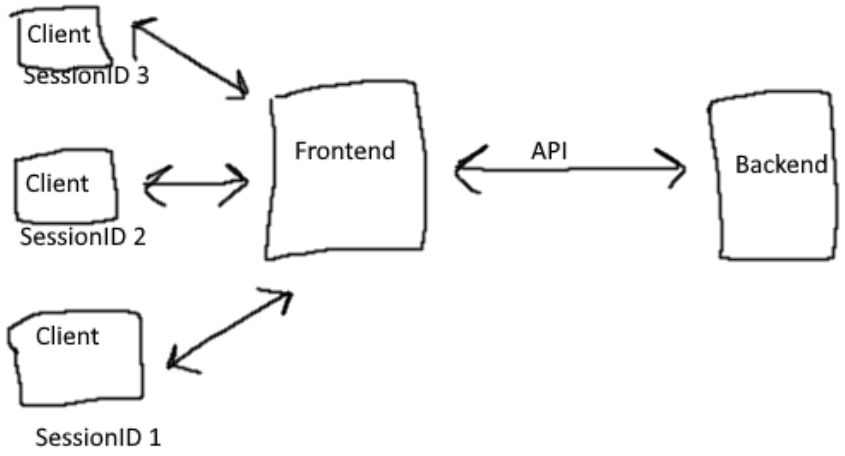
- **Virtual Machines (VMs)** – Used for running isolated environments.
- **Serverless Computing** – Automatically scales without managing servers.

Application Layer

How does the Backend and the frontend communicate?

📌 **Example Flow (REST API Communication):**

1. **User Action** – A user submits a form on the frontend (e.g., signing up).
2. **Frontend Request** – The frontend **sends a POST request** to the backend with the form data.
3. **Backend Processing** – The backend **validates the data, updates the database**, and processes business logic.
4. **Backend Response** – The backend sends a **success/failure response** back to the frontend.
5. **Frontend Update** – The frontend updates the **UI dynamically** based on the response (e.g., "Registration Successful!").



Application Layer: APIs

Application Programming Interface (API)

What is an API?

API is a tool that lets two different software programs (or "applications") talk to each other and share things, like information or features.

- **API (Application Programming Interface)** is a **software interface** that allows two or more applications to communicate with each other.
- It acts as a **bridge** between different software components, enabling seamless integration. It's a bridge that connects apps smoothly.
- APIs **abstract away complexities**, allowing developers to interact with services without knowing the internal implementation.
  It hides the hard stuff so developers don't need to figure it out
- It defines a **contract of services**, specifying how applications should request and exchange data

For example, when you use a weather app, it asks a weather API for data (like temperature) and shows it to you—without you needing to understand how the weather service works behind the scenes.

Application Layer: APIs

Benefits of APIs?

Standardized Communication

- Uses standard protocols like **HTTP/HTTPS, WebSockets, gRPC**.
- Responses are typically in **JSON** or **XML** format.

Encapsulation & Abstraction

- **Hides complex implementation details**, exposing only necessary functions and data..
- Developers don’t need to understand database queries or backend logic.

Reusability & Efficiency

- APIs allow applications to **reuse functionality, reducing redundant code**.
- ***Example***: A payment gateway API (like Stripe) can be integrated into multiple applications without writing new payment logic.

Application Layer: APIs

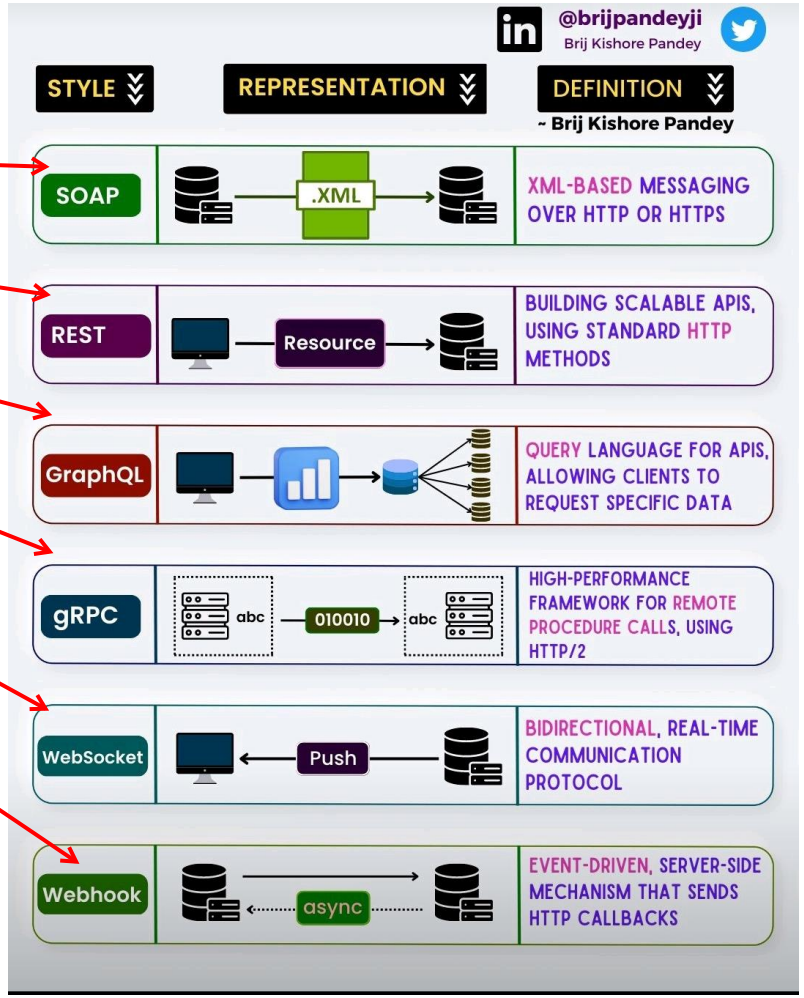
Different Types of APIs

REST API (Representational State Transfer)

- ✅ Uses **HTTP methods** (GET, POST, PUT, DELETE) for communication.
- ✅ Stateless and follows **resource-based architecture**.
- ✅ Responses are typically in **JSON or XML**.
- 📌 **Commonly Used For:** Web and mobile applications, cloud services.

SOAP API (Simple Object Access Protocol)

- ✅ Uses **XML-based messaging** with strict standards.
- ✅ Supports **ACID transactions and high-security protocols**.
- ✅ More complex than REST but ideal for **enterprise-level applications**.
- 📌 **Commonly Used For:** Banking, financial services, and legacy systems.



Application Layer: APIs

RPC (Remote Procedure Call)

- ✅ Allows one system to execute **a function/method** on another system.
- ✅ Can use **JSON-RPC** or **XML-RPC** for structured messaging.
- ✅ **Faster than REST** but tightly coupled to specific implementations.
- 📌 **Commonly Used For:** Microservices and distributed computing.

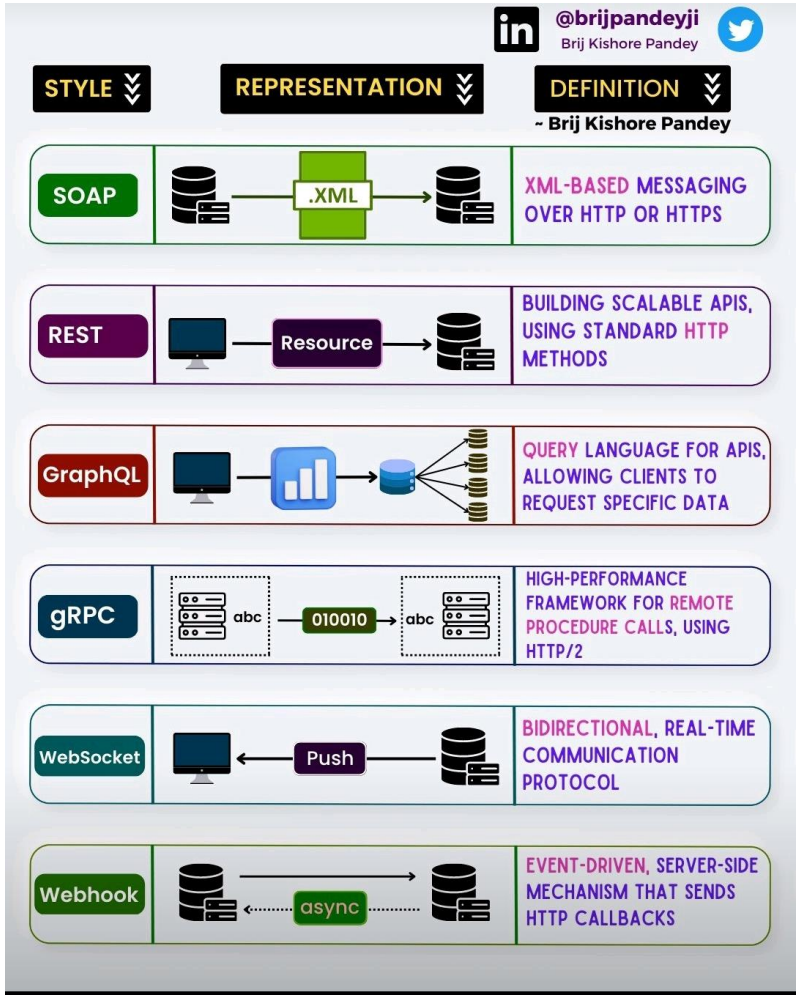
WebSockets API

- ✅ Provides **real-time, bidirectional communication** between client and server.
- ✅ Unlike REST, it **maintains an open connection** for continuous data flow.
- ✅ Ideal for **high-speed, interactive applications**.
- 📌 **Commonly Used For:** Chat applications, gaming, live updates.

GraphQL API

- ✅ A flexible **query language for APIs**, allowing clients to request specific data.
- ✅ Reduces **over-fetching and under-fetching** of data.
- ✅ **More efficient** than REST for complex applications.
- 📌 **Commonly Used For:** Social media, e-commerce, and data-heavy applications.

Getting more data than you need
Not getting enough data



Architecting the Application Layer

Some Well-Known Examples of Application Layer Architectures

- Monolithic Architecture
- Microservices Architecture
- Service-Oriented Architecture (SOA)

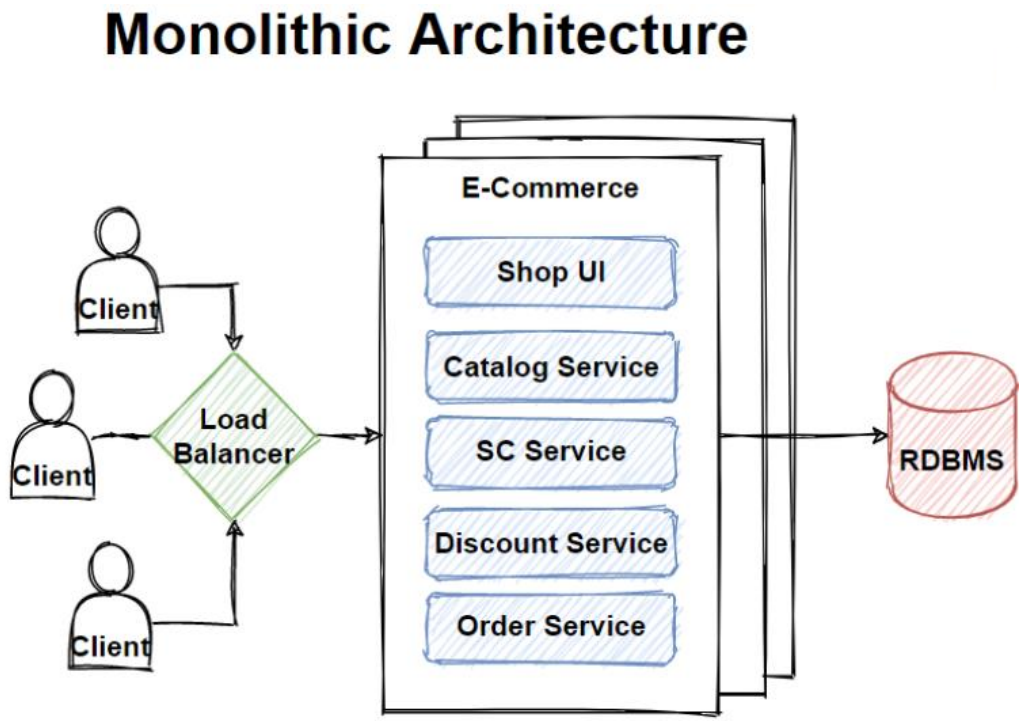
📌 **[Read] What is SOA?**

Architecting the Application Layer: Monolithic

◆ **What is Monolithic Architecture?**

- A **traditional software architecture** where the entire application is built as **a single unit**.
- All components, including **UI, business logic, and database operations**, are tightly coupled.
- The application is **deployed as one entity**, meaning changes require redeploying the whole system

Architecting the Application Layer: Monolithic



Source: https://medium.com/design-microservices-architecture-with-patterns/monolithic-architecture-is-still-worth-at-2021-98bfc112dc24

Architecting the Application Layer: Monolithic

Drawbacks in Monolithic Architecture?

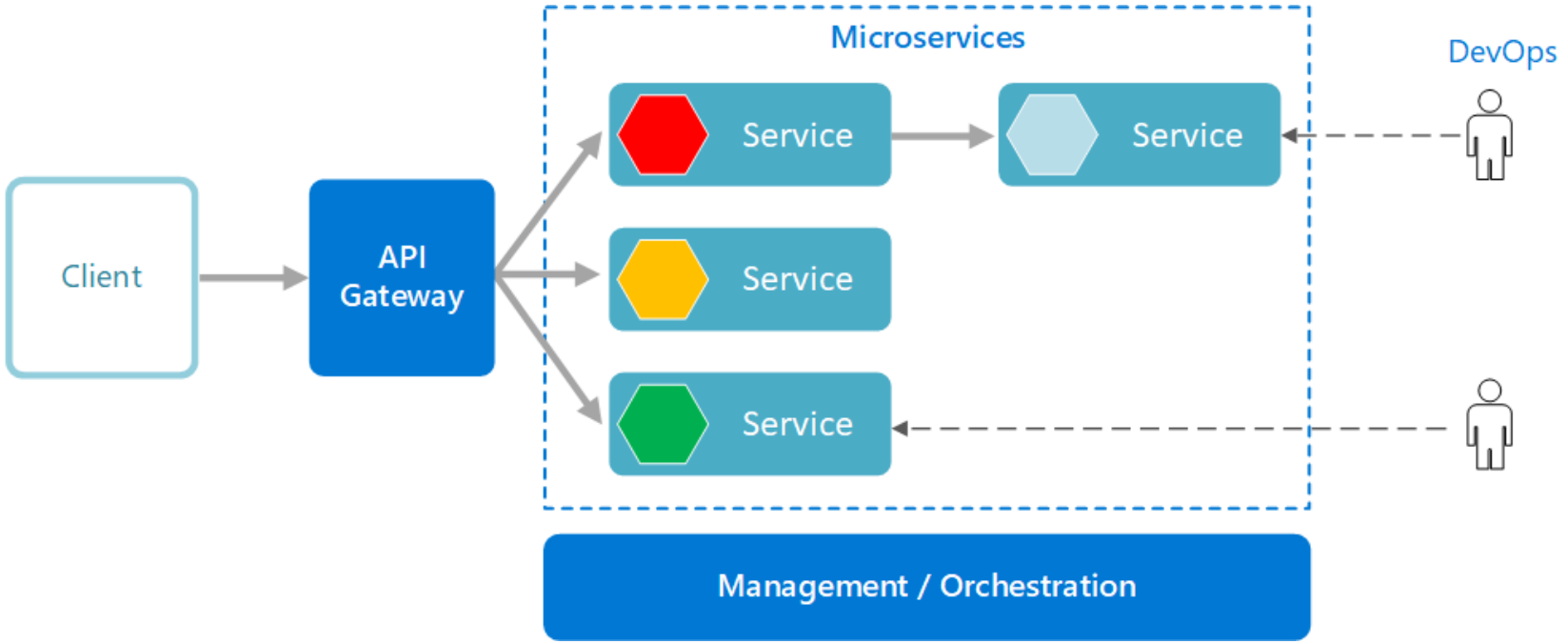
- **Scalability Challenges** – Cannot scale individual components independently.
- **Slower Development Speed** – As the codebase grows, adding new features becomes complex.
- **High Risk of Failure** – A failure in one module **can crash the entire application**.
- **Difficult Technology Upgrades** – Any change in the framework or language affects the whole system.
- **Deployment Issues** – Even a **small update** requires redeploying the **entire** application.
- **Limited Flexibility** – Developers are restricted to **one technology stack** for the whole application.

Architecting the Application Layer: Microservices

What is Microservices Architecture?

- A **modern architectural approach** where an application is divided into **small, independent, and** loosely coupled **services**.
- Each **microservice handles a specific business function** and communicates via **APIs (REST, gRPC, or messaging queues)**.
- Services can be **developed, deployed, and scaled independently**, making it ideal for large-scale applications.

Architecting the Application Layer: Microservices



- Source: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

Architecting the Application Layer: Microservices

Benefits of Microservices Architecture

🚀 **Scalability** – Each service can be scaled **independently**, optimizing resource usage.

🛠️ **Flexibility** – Allows using **different technologies and programming languages** for different services.

🔄 **Faster Development & Deployment** – Small teams can **develop and deploy microservices separately**, enabling **continuous delivery**.

⚡ **Improved Fault Isolation** – A failure in one service **does not affect the entire system**.

Architecting the Application Layer: Microservices

🔧 **Easier Maintenance** – Since services are modular, updates and bug fixes **are simpler and faster**.

🔗 **Loosely Coupled Services** – Microservices interact **through APIs**, reducing dependencies and **allowing independent changes**.

📡 **Better Performance** – Services can be deployed **closer to users** for improved speed and efficiency.

🔄 **Supports DevOps & CI/CD** – Works well with **automated deployment pipelines** for rapid updates.

Architecting the Application Layer: Microservices

⚠ Drawbacks of Microservices Architecture

High Infrastructure Costs

- Requires more servers, databases, and network resources.

Debugging & Troubleshooting Challenges

- Hard to trace issues across multiple services.

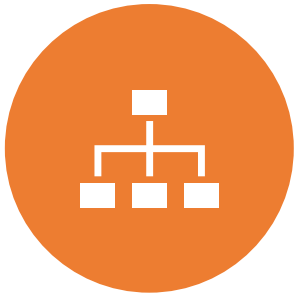
Complex Deployment & Maintenance

- Requires strong DevOps & CI/CD automation.

Security & Data Management Concerns

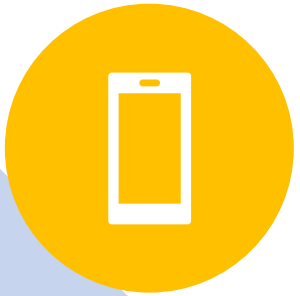
- Increased attack surface due to multiple exposed APIs.
- Data consistency challenges across multiple databases.

Introduction to REST API

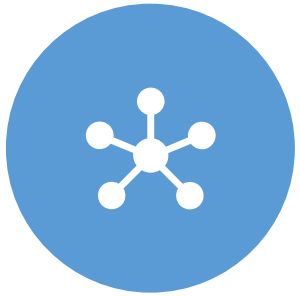


REST (Representational State Transfer) is an architectural style for building web services

REST APIs allow clients to interact with server-side resources over the internet using HTTP methods (GET, POST, PUT, DELETE)



REST APIs are widely used for building web applications and mobile apps



Why use REST API? It provides a simple and standardized way to build scalable and interoperable systems.

REST API Design Principles

✅ **Use Proper HTTP Methods**

- **GET** → Retrieve data (e.g., /users)
- **POST** → Create a new resource (e.g., /users)
- **PUT** → Update an existing resource (e.g., /users/123)
- **DELETE** → Remove a resource (e.g., /users/123)

✅ **Use Meaningful URIs**

- Keep URIs **resource-oriented** (e.g., /products/123, not ~~*/getProduct?id=123*~~).
- Use **nouns instead of verbs**.

✅ **Return Correct HTTP Status Codes**

- **200 OK** – Successful request
- **201 Created** – Resource created successfully
- **404 Not Found** – Resource not found
- **500 Internal Server Error** – Unexpected failure

REST API Design Principles

✅ Ensure a Consistent API Interface

- Use **structured JSON responses** for readability.
- Example response for GET /users/123

```
{
  "id": 123,
  "name": "John Doe",
  "email": "john.doe@example.com"
}
```

Hypermedia As The Engine Of Application State.

✅ Implement Hypermedia (HATEOAS)

REST APIs that makes them more "self-describing" by including links in the response.
hese links tell the app what it can do next or where to get more data.

- Provide **links** in responses for easy API navigation.

```
{
  "id": 123,
  "name": "John Doe",
  "links": {
    "self": "/users/123",
    "orders": "/users/123/orders"
  }
}
```

REST API Request Format

A REST API request typically consists of:

HTTP Method

- Defines the type of operation being performed:
  - **GET** → Retrieve data
  - **POST** → Create a new resource
  - **PUT** → Update an existing resource
  - **DELETE** → Remove a resource

URI (Uniform Resource Identifier)

- Specifies the location of the resource being accessed.
  - Example:
  - /users → Retrieves all users
  - /products/123 → Retrieves product with ID 123

REST API Request Format

A REST API request typically consists of:

Headers

- Contain **metadata** about the request.
- Examples**:**
  - **Content-Type**: application/json → Defines data format.
  - **Authorization**: Bearer <token> → Used for authentication.URI (Uniform Resource Identifier)

Body (Optional)

- Contains data sent with the request (mostly for POST and PUT).
- Typically formatted as JSON or XML.
- Example POST request body:

```
{
   "name": "John Doe",
   "email": "john.doe@example.com",
   "password": "securepassword123"
}
```

REST API Response Format

A REST API response typically consists of:

HTTP Status Code

- Indicates the **success or failure** of the request.
- Common status codes:
  - **200 OK** → Request was successful.
  - **201 Created** → A new resource was successfully created.
  - **400 Bad Request** → Invalid input from the client.
  - **404 Not Found** → Requested resource does not exist.
  - **500 Internal Server Error** → Unexpected issue on the server.

Headers

- Contain **metadata** about the response.
- Common headers:
  - **Content-Type**: application/json → Defines the response format.
  - **Cache-Control**: no-cache → Instructs the client not to cache the response.
  - **Authorization**: Bearer abc123token → Ensures secure access to protected resources.

REST API Response Format

A REST API response typically consists of:

Body (Response Payload)

- Contains the **actual data** being sent in the response.
- Usually formatted in **JSON** or **XML**.
- Example response (JSON format):

```
{
   "id": 123,
   "name": "John Doe",
   "email": "john.doe@example.com",
   "message": "User details retrieved successfully."
}
```

HTTP Request Message

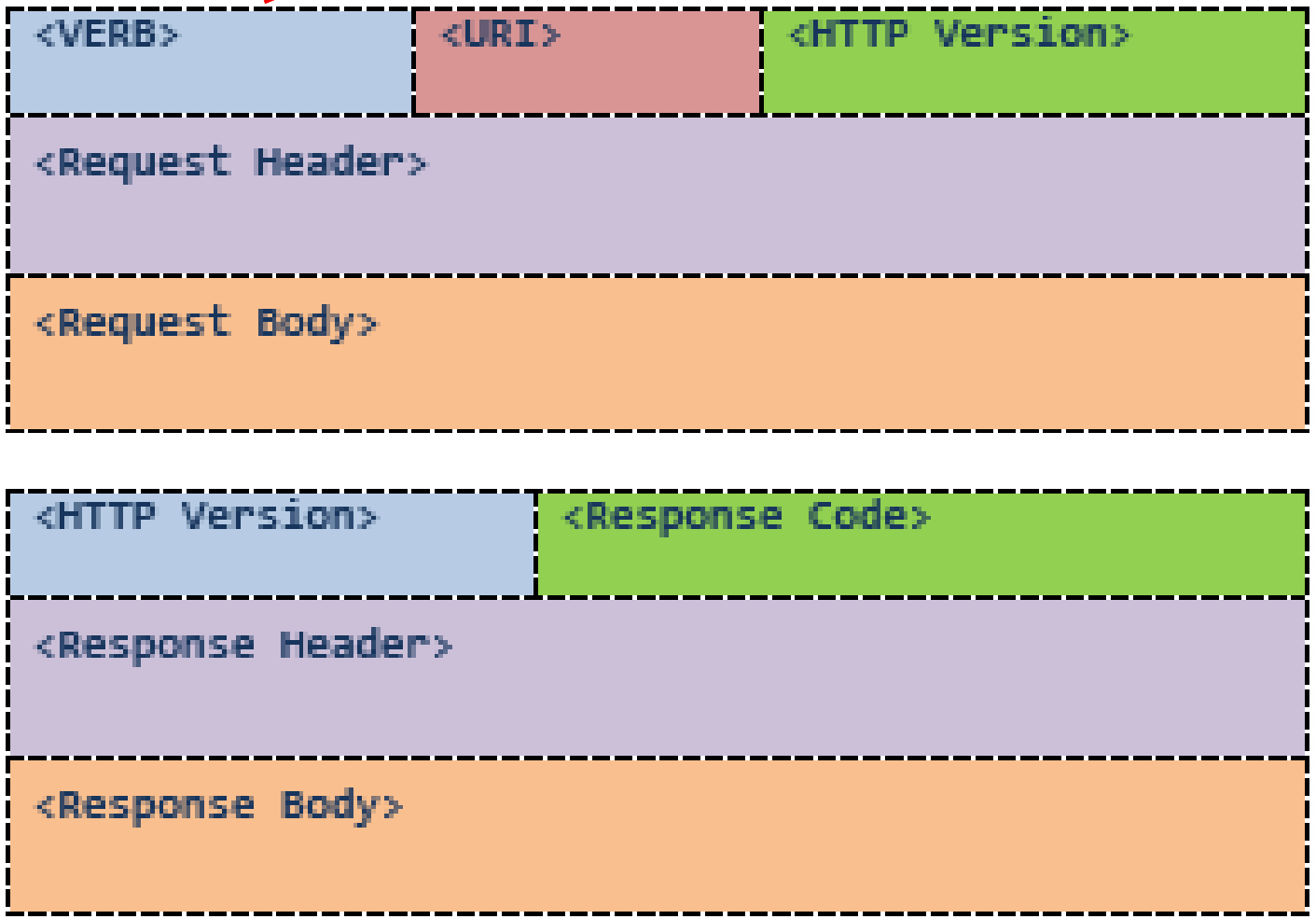
```
POST /users HTTP/1.1
Host: example.com
Content-Type: application/json
Content-Length: 27

{"name": "Jane", "age": 25}
```

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 90

{
  "id": 456,
  "title": "Cool Story",
  "links": [
    {"rel": "author", "href": "/articles/456/author"}
  ]
}
```



HTTP Response Message

HTTP codes

- HTTP status codes are **3-digit numbers** that indicate the status of a **client's request to a server**.
- The **first digit** defines the **class of response**.
- The **last two digits** provide **specific details** about the response.
  - 1xx Informational
  - 2xx Success
  - 3xx Redirection
  - 4xx Client Error
  - 5xx Server Error

Commonly used HTTP codes

✅ **200 OK (Success)**

- The request was **successful**, and the server has returned the requested data.
- Example:

  GET /users/123 → Returns user details with status 200 OK.

🔀 **301 Moved Permanently (Redirection)**

- The requested resource **has been permanently moved** to a new location.
- The client should update its **URL** to reflect this change.
- Example:

  GET /old-page → Redirects to /new-page with status 301 Moved Permanently.

Commonly used HTTP codes

❌ 404 Not Found (Client Error)

- 🔷 The requested resource does not exist on the server.
- 🔷 Happens when a URL is incorrect or outdated.
- 🔷 Example:

  GET /products/999 (Product ID does not exist) → Returns 404 Not Found.

⚠ 500 Internal Server Error (Server Error)

- 🔷 The server encountered an unexpected issue while processing the request.
- 🔷 The client should retry later as this is a server-side issue.
- 🔷 Example:

  POST /checkout (Database failure) → Returns 500 Internal Server Error.

HTTP Code Examples

200 - OK *

201 - Created *

202 - Accepted

204 - No content *

301 - Moved permanently

302 - Found

304 - Not modified *

400 - Bad request *

401 - Unauthorized *

403 - Forbidden * unauthentication issue

404 - Not found *

HTTP Code Examples

405 - Method not allowed

409 - Conflict *

412 - Precondition failed

418 – An interesting code check  418 I'm a Teapot

It's not a serious error or success code

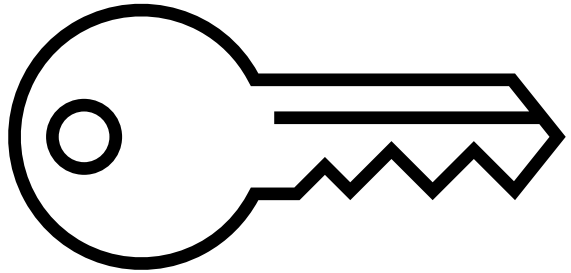
it's a joke from an April Fool's prank!It means the server refuses to process the request because it's a "teapot," not something that can handle the task

500 - Internal server error *

502 - Bad gateway

503 - Service unavailable

REST API Authentication



🛠 What is Authentication?

- Authentication is the process of verifying the identity of a client making a request to a REST API.
- It ensures that only authorized users or applications can access protected resources.

🔷 Common REST API Authentication Mechanisms

✅ Basic Authentication

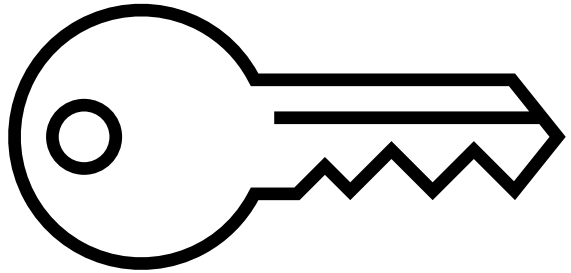
The client sends a **base64-encoded** **username and password** in the Authorization header.

Example:

Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=

🔴 **Less secure** as credentials are sent in every request.

REST API Authentication



✅ Token-Based Authentication

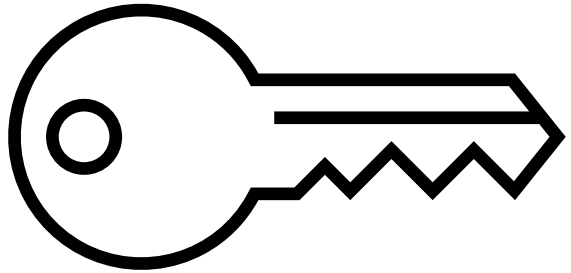
The client sends a token (e.g., JWT - JSON Web Token) in the Authorization header.

Example:

Authorization: Bearer <JWT_TOKEN>

✅ **More secure** as credentials are exchanged only once to generate a token.

REST API Authentication



✅ OAuth 2.0 (Authorization Protocol)

- Allows a user to **grant a third-party application** access to their resources without sharing credentials.
- Uses **Access Tokens** to authenticate requests.
- Commonly used in **Google**, **Facebook**, **and GitHub APIs.**

Example flow:

1. User logs in via OAuth provider (Google, Facebook, etc.).
2. Provider issues an Access Token.
3. Client includes the token in API requests:

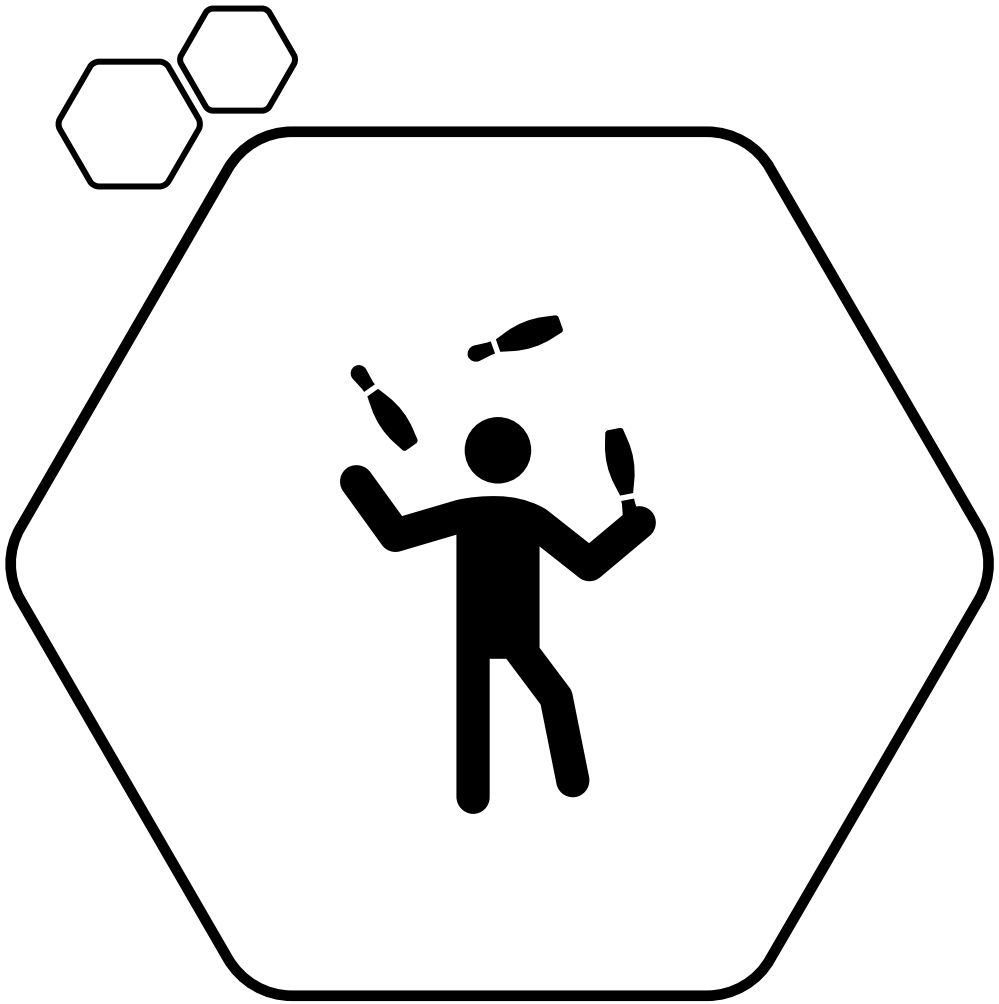
```
Authorization: Bearer <OAUTH_TOKEN>
```

REST API Versioning

- REST APIs **evolve over time**, and new versions may introduce **changes that are not backward compatible**.
- API versioning helps maintain **stability for existing users** while allowing improvements and updates.

Versioning can be done by:

- **Using a version number in the URI** (e.g., /v1/users)
- **Using a version number in the Accept header** (e.g., Accept: application/json; version=1.0)
- **Using content negotiation** to select the appropriate version based on the client's request.(*The server **dynamically selects** the correct version based on the request*.)



REST API Best Practices

Some best practices to follow when designing and implementing REST APIs include:

- Keeping The API Simple And Consistent
- Using Descriptive And Meaningful Uris
- Providing Clear Documentation And Examples
- Avoiding Breaking Changes In New Versions
- Using Caching And Compression To Improve Performance
- Monitoring And Analyzing Usage Data To Identify Issues And Optimize The API


Express
JS

Introduction to Express JS

doesn't force you into one way of doing things—you have freedom to structure your app how you like.

- Express JS is a fast, unopinionated, and minimalist web framework for Node.js.

It gives you the basics and lets you add what you need, keeping it simple.

- It was created in 2010 by TJ HolowaychuK and is now maintained by the Node.js Foundation.
- Express JS is widely used for building web applications and APIs.

Why use Express JS?

It provides a simple and flexible way to handle HTTP requests and responses, making it easy to build scalable and maintainable applications.

Setting up an Express JS application

🛠 Getting Started with Express.js

To use **Express.js**, you first need to install it and set up your project.

🔷 Step 1: Install Express.js

Express.js is installed using npm (Node Package Manager).

📌 Run the following command:

```
npm install express
```

✅ Installs Express.js and saves it as a dependency in package.json.

🔷 Step 2: Create an Express.js Application

📌 Option 1: Using the Express Generator (Automated Setup)

```
npx express-generator my-app
cd my-app
npm install
```

✅ Automatically creates a structured project with essential files.

Setting up an Express JS application

📌 Option 2: Manually Creating the Project

```
mkdir my-app && cd my-app
npm init -y
npm install express --save
touch app.js
```

✅ Manually sets up the project with Express.js.

🔷 **Step 3: Understanding Key Files**

📌 Important files in an Express.js application:

✓ app.js (or index.js) → Main file where the server is created.

✓ package.json → Manages dependencies and scripts.

✓ node_modules/ → Stores installed npm packages.

Setting up an Express JS application

🔷 Step 4: Start the Express.js Server

📌 Run the server using one of the following commands:

```
npm start
```

✅ Starts the server, making the app accessible at http://localhost:3000.

Routing in Express JS

🛠 What is Routing?

Routing is the process of **matching a URL pattern** to a **specific controller function** that handles the request.

It determines **how an application responds** to different HTTP requests.

Routing in Express JS

🔷 Defining Routes in Express.js

In Express.js, routes are defined using **HTTP methods** such as:

📌 Example: Basic Routes in Express.js

```
// Define Routes
app.get('/users', (req, res) => {
    res.send('Retrieving all users');
});

app.post('/users', (req, res) => {
    res.send('Creating a new user');
});

app.put('/users/:id', (req, res) => {
    res.send(`Updating user with ID ${req.params.id}`);
});

app.delete('/users/:id', (req, res) => {
    res.send(`Deleting user with ID ${req.params.id}`);
});
```

Routing in Express JS

- **Using Route Parameters**

✅ Route parameters allow dynamic values in URLs.

✅ Example: /users/:id → req.params.id extracts the user ID.

📌 Example: Using Route Parameters

```
app.get('/users/:id', (req, res) => {
    res.send(`User ID: ${req.params.id}`);
});
```

- Request: GET /users/123
- Response: "User ID: 123"

Routing in Express JS

🔷 Using Regular Expressions in Routes

✅ You can use regular expressions to match complex URL patterns.

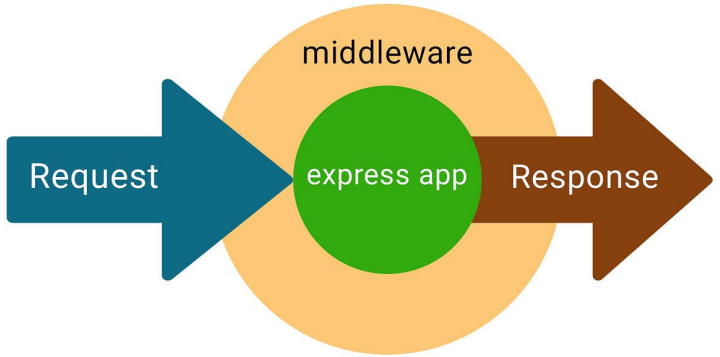
📌 Example: Matching Numeric User IDs Only

```
app.get('/users/:id([0-9]+)', (req, res) => {
  res.send(`User ID: ${req.params.id}`);
});
```

🔷 Only matches numeric IDs (/users/123 ✅, /users/abc ❌).

numeric only

Middleware in Express JS



Middleware is a **function that sits between** the **client request** and the **server response**. It can **modify** the request, response, or terminate the request cycle.

- Express JS has **built-in middleware** functions for handling requests, parsing data, and serving static files
- You can use **third-party middleware** functions for tasks like logging, authentication, and caching
- You can **create your own middleware** functions to handle specific tasks or modify the request/response as needed

express.json(): Parses incoming requests with JSON payloads.
express.urlencoded(): Parses incoming requests with URL-encoded payloads.
express.static(): Serves static files from a directory.
express.Router(): Modularizes routes using a router instance.

Handling Errors In Express JS

Error handling ensures that **unexpected issues** in an application are properly managed, improving **stability and user experience**.

🔷 How Express.js Handles Errors?

- ✅ Built-in Error Handling – Automatically catches and responds to errors.
- ✅ Custom Error Middleware – Allows structured and meaningful error messages.

🔷 Common Types of Errors

- ❌ 404 Not Found – The requested resource doesn't exist.
- ⚠ 500 Internal Server Error – Server-side issue or unexpected failure.
- ⚙ User-Defined Errors – Custom application-specific errors (e.g., authentication failures).

🔷 Best Practices

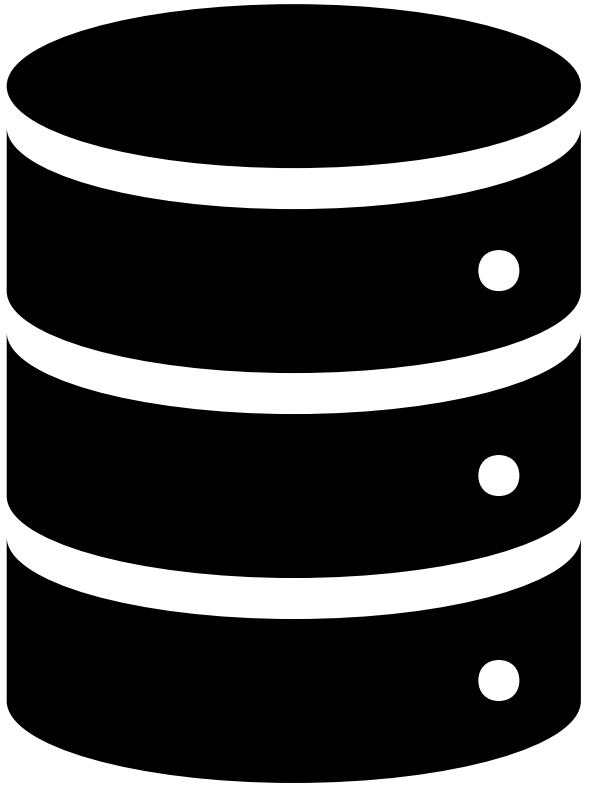
- ✓ Log errors for debugging.
- ✓ Provide clear & consistent error messages.
- ✓ Use proper HTTP status codes (404, 500, etc.).

Templating Engines In Express JS

- Templating engines are used to generate HTML pages dynamically
- Express JS supports several popular templating engines, including Pug (formerly Jade), EJS (Embedded JavaScript), and Handlebars
- You can install a templating engine using npm and then configure it in your Express JS application
- Templating engines allow you to create reusable templates and pass data to them from your controllers

Working With Databases In Express JS

- Express JS can work with different types of databases, including SQL databases (e.g., MySQL, PostgreSQL) and NoSQL databases (e.g., MongoDB)
- You can connect to a database using a database driver or an ORM (Object-Relational Mapping) library like Sequelize or Mongoose
- Querying the database is done using SQL or a database-specific query language (e.g., MongoDB query syntax)
- ORM libraries provide a higher-level interface for working with databases and can simplify the code for common tasks like creating, reading, updating, and deleting records



Restful Apis With Express JS

- A RESTful API is an API that follows the REST (Representational State Transfer) architecture style.
- Express JS is well-suited for building RESTful APIs because of its routing and middleware capabilities.
- To create a RESTful API, you define routes that correspond to the different HTTP methods (GET, POST, PUT, DELETE) and the resources you want to expose.
- You can use middleware functions to handle tasks like input validation, authentication, and rate limiting.
- RESTful APIs should follow certain principles, such as using HTTP status codes to indicate success or failure, using meaningful URIs to identify resources, and providing a consistent interface for interacting with the API.

Thank you!

References

- https://www.npmjs.com/package/express-basic-auth
- https://www.npmjs.com/package/jsonwebtoken
- https://expressjs.com/en/starter/installing.html
- https://www.javatpoint.com/expressjs-template#:~:text=A%20template%20engine%20facilitates%20you,to%20design%20HTML%20pages%20easily.