

Chapter 17: MERN REST API Development

17.1 MERN Architecture

17.1.1 What is MERN Stack?

- * MERN is a popular JavaScript stack used for building modern web applications.
- * It stands for **MongoDB**, **Express.js**, **React.js** and **Node.js**.
- * It allows developers to build **full-stack** applications using JavaScript on both frontend and backend.
- * MERN follows a component-based and **API-driven** architecture.

MERN Stack



MongoDB
Database



Express.js
Backend
Framework



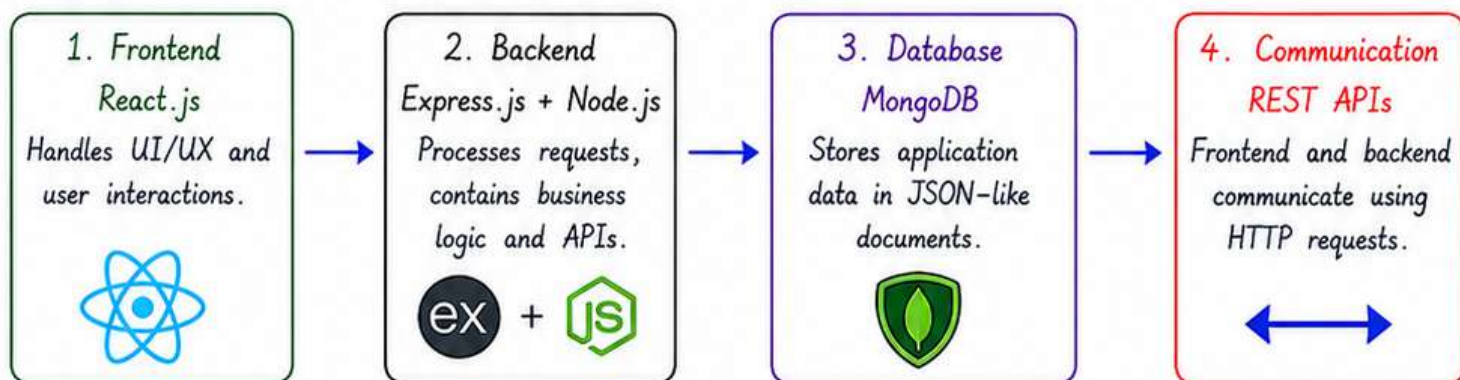
React.js
Frontend
Library



Node.js
Runtime
Environment

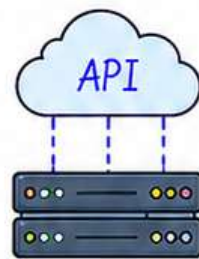
17.1.2 MERN Architecture Overview

MERN architecture consists of 4 main layers working together.



Key Points:

- ✓ MERN stack uses **JavaScript** for both frontend and backend.
- ✓ React.js builds **dynamic** and **responsive** user interfaces.
- ✓ Express.js and Node.js handle the **server-side logic** and **API** development.
- ✓ MongoDB stores data in **flexible**, **scalable** and **JSON-like** format.
- ✓ The frontend communicates with the backend using **RESTful APIs**.



17.1.3 Benefits of MERN Architecture



Full JavaScript Stack - One language for entire development.



Scalable - Can handle small projects to large applications.



Active Community - Huge support and resources available.



Easy Integration - All components work seamlessly together.



High Performance - Built for speed and efficiency.



JSON-based - Easy data exchange and flexibility.

Key Takeaway:



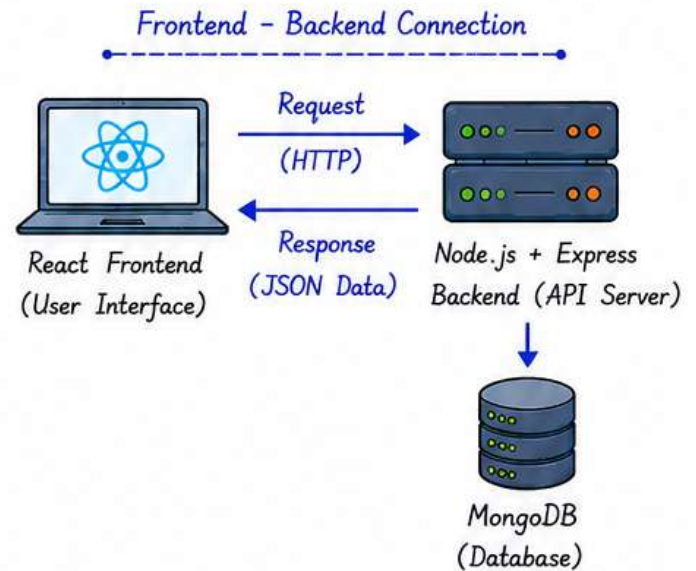
MERN architecture provides a **powerful** and **efficient** way to build modern web applications using **JavaScript** across the entire stack with **REST API** communication.



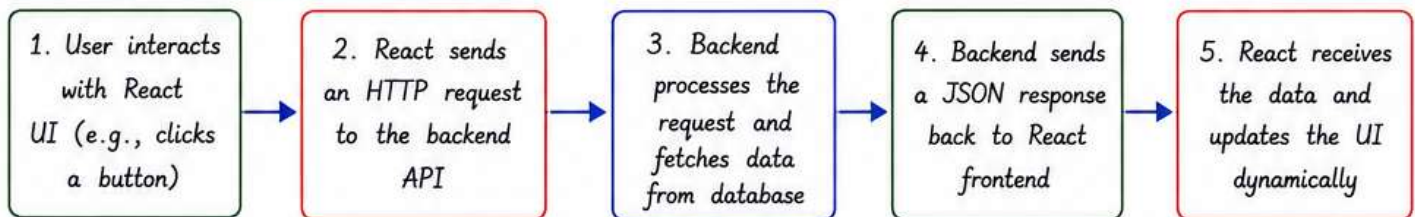
17.2 Frontend-Backend Integration

17.2.1 What is Frontend-Backend Integration?

- * Frontend-Backend integration is the process of connecting a **React** frontend with a **Node.js + Express** backend using **REST APIs**.
- * It allows the frontend to **send requests** to the backend and **receive data**.
- * The data is usually exchanged in **JSON** format.
- * This integration makes the web application **dynamic** and **interactive**.

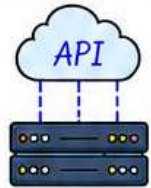


17.2.2 How Integration Works?



Key Points:

- ✓ **React** (frontend) communicates with **Node.js + Express** (backend) using **HTTP** requests.
- ✓ The most common data format used is **JSON**.
- ✓ APIs act as a **bridge** between frontend and database.
- ✓ This separation improves scalability, **maintainability** and **performance**.



17.2.3 Example: Fetching Data from Backend

Frontend (React) - Sending Request

```
import React, { useEffect, useState } from 'react';
function UserList() {
  const [users, setUsers] = useState([]);
  useEffect(() => {
    fetch('http://localhost:5000/api/users')
      .then(res => res.json())
      .then(data => setUsers(data))
      .catch(err => console.log(err));
  }, []);
  return (
    <div>{/* display users */</div>
  );
}
```

Request
(GET)

Response
(JSON)

Backend (Node.js + Express) - API Endpoint

```
const express = require('express');
const app = express();

app.get('/api/users', async (req, res) => {
  const users = await User.find();
  res.json(users);
});

app.listen(5000, () => {
  console.log('Server running on port 5000');
});
```

17.2.4 Best Practices



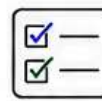
Use **HTTPS** for secure communication.



Handle errors properly.



Show loading state while fetching data.



Validate data on the backend.



Always use **JSON** for data exchange.

Key Takeaway:



Frontend-backend integration allows **React** applications to communicate with the backend using **REST APIs**, enabling **dynamic data handling** and a **smooth user experience**.

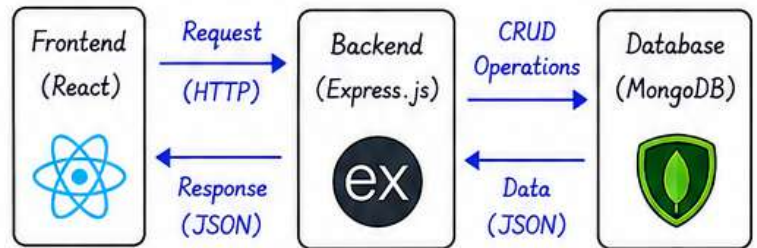


17.3 Express + MongoDB

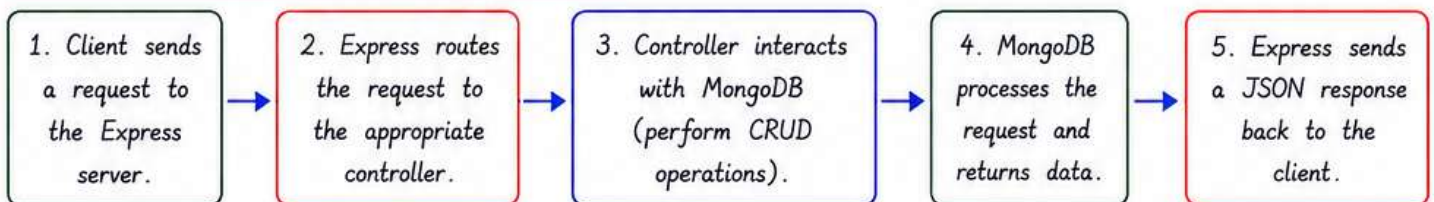
17.3.1 What is Express + MongoDB?

- * Express.js is a backend framework for Node.js used to build **REST APIs** and web applications.
- * **MongoDB** is a **NoSQL** database used to store data in **JSON-like** documents.
- * Together, Express and MongoDB provide a powerful combination for building **scalable** and **efficient** backend systems.

Express + MongoDB Architecture



17.3.2 How Express + MongoDB Works?



Key Points:

- ✓ Express handles **routing**, **middleware** and **business logic**.
- ✓ MongoDB stores data in **flexible**, **schema-less** collections.
- ✓ **Mongoose** is used to connect and interact with MongoDB in Express apps.
- ✓ Together, they enable fast development of **RESTful APIs**.



17.3.3 Example: Simple CRUD API (User)

Express.js API (server.js)

```
const express = require('express');
const mongoose = require('mongoose');
const app = express();
app.use(express.json());

mongoose.connect('mongodb://127.0.0.1:27017/mernapp')
  .then(() => console.log('MongoDB Connected'))
  .catch(err => console.log(err));

const userSchema = new mongoose.Schema({
  name: String,
  email: String
});
const User = mongoose.model('User', userSchema);

// GET: Get all users
app.get('/api/users', async (req, res) => {
  const users = await User.find();
  res.json(users);
});

app.listen(5000, () => console.log('Server running on port 5000'));
```

Example User Document (MongoDB)

```
{
  "_id": ObjectId("663f7d2b5e1b2c001f4a1c88"),
  "name": "Bhavitha",
  "email": "bhavi@example.com",
  "__v": 0
}
```

Request
(HTTP)

Response
(JSON)

Common CRUD Operations

Operation	Method	Endpoint	Description
Create	POST	/api/users	Create a new user
Read (All)	GET	/api/users	Get all users
Read (By ID)	GET	/api/users/:id	Get user by ID
Update	PUT	/api/users/:id	Update user by ID
Delete	DELETE	/api/users/:id	Delete user by ID

17.3.4 Benefits



Fast development with Node.js and Express.



Flexible and scalable data storage with MongoDB.



Easy to build RESTful APIs.



Well-suited for modern web applications.



Secure and reliable for handling data.

Key Takeaway:



Express.js and MongoDB together provide a powerful backend solution for building **REST APIs**. Express handles requests and MongoDB stores data in a flexible, JSON-like format, making the MERN stack **efficient** and **scalable**.

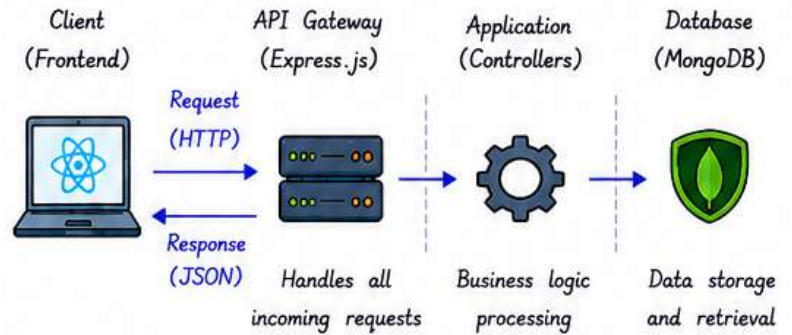


17.4 API Architecture

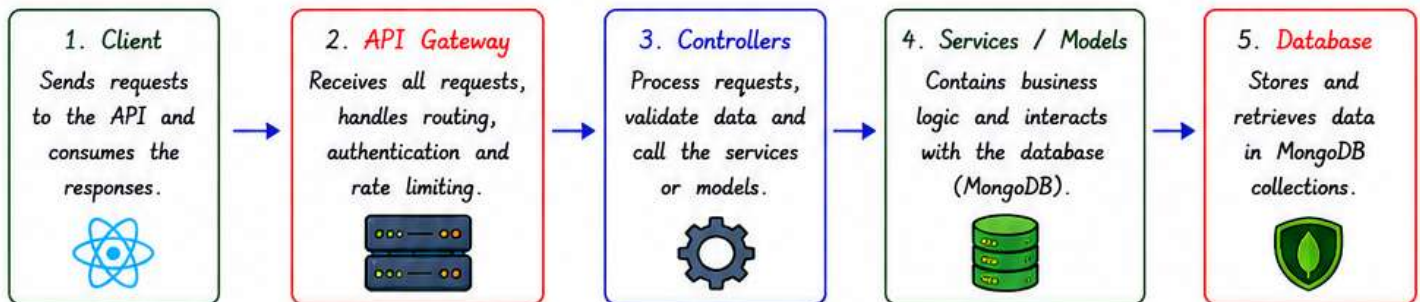
17.4.1 What is API Architecture?

- * API Architecture defines how APIs are designed, structured and communicated in a system.
- * It describes the components, their interactions and the data flow.
- * A well-designed API architecture ensures **scalability, security, maintainability** and **performance**.
- * In MERN stack, APIs are built using **Node.js + Express** and data is stored in **MongoDB**.

High-Level API Architecture



17.4.2 Components of API Architecture

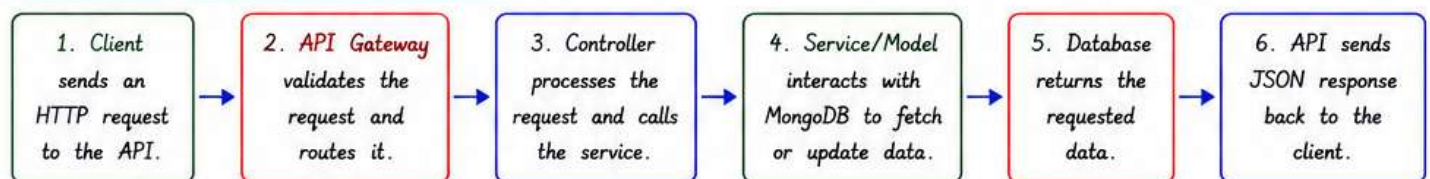


Key Points:

- ✓ API architecture follows a **layered approach** for better separation of concerns.
- ✓ HTTP/HTTPS is used for communication between client and server.
- ✓ Data is exchanged in **JSON** format.
- ✓ Each layer has a specific responsibility.
- ✓ This architecture ensures **scalability, security** and **high performance**.



17.4.3 API Request-Response Flow



17.4.4 Example API Endpoints (User Management)

Method	Endpoint	Description	Request Body	Response
GET	/api/users	Get all users	-	List of users (JSON)
GET	/api/users/:id	Get user by ID	-	User object (JSON)
POST	/api/users	Create a new user	{ name, email, password }	New user object (JSON)
PUT	/api/users/:id	Update user by ID	{ name, email, password }	Updated user object (JSON)
DELETE	/api/users/:id	Delete user by ID	-	Success message (JSON)

17.4.5 Best Practices

- Use **HTTPS** for secure communication.
- Follow **RESTful** API conventions.
- Keep responses consistent.
- Implement authentication and authorization.
- Handle errors and validation properly.
- Optimise database queries.

Key Takeaway:



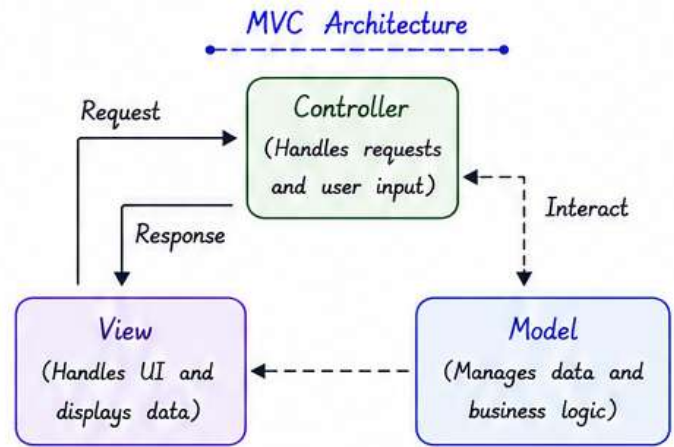
A well-structured API architecture in MERN stack ensures smooth communication between client and server, **secure** data handling and a **scalable** application.



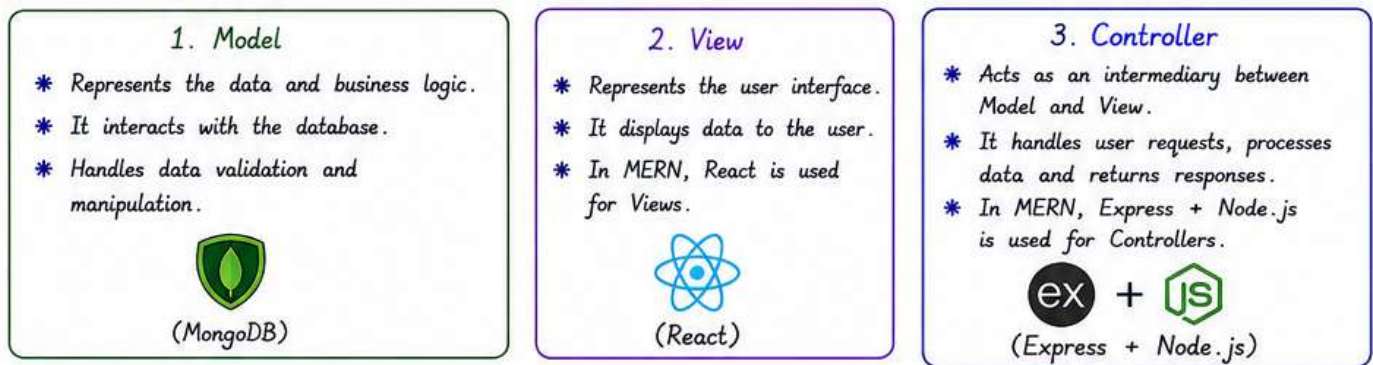
17.5 MVC Pattern

17.5.1 What is MVC Pattern?

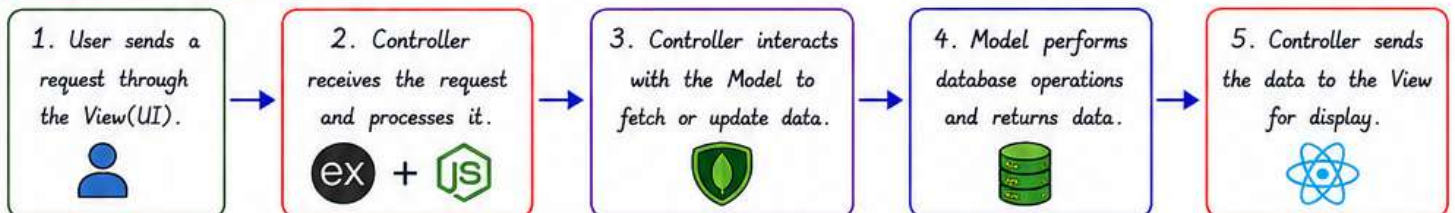
- * MVC (Model-View-Controller) is a software design pattern used to **separate** an application into three interconnected components.
- * It helps in organizing code, improving **maintainability** and promoting **reusability**.
- * In the MERN stack, MVC is commonly used in the backend (**Node.js** + **Express**).
- * It separates **business logic**, **data** and **user interface** logic.



17.5.2 MVC Components

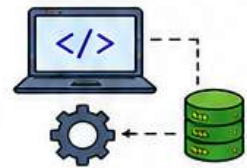


17.5.3 How MVC Works?



Key Points:

- ✓ MVC separates an application into Model, View and Controller.
- ✓ It improves code organization and maintainability.
- ✓ Model handles data, View handles UI and Controller handles logic.
- ✓ In MERN stack, React (View), Express (Controller), MongoDB (Model).
- ✓ MVC makes applications easier to develop and test.



17.5.4 Example in MERN (User List)

MVC Component	Technology (MERN)	Responsibility	Example
Model	MongoDB + Mongoose	Stores and manages user data	User Schema, User Model
View	React.js	Displays user list in UI	UserList.jsx component
Controller	Express.js + Node.js	Handles API requests and responses	userController.js

17.5.5 Benefits of MVC Pattern



Better code organization.



Easy to maintain and update.



Multiple developers can work easily.



Easier testing and debugging.



Promotes reusability and scalability.

Key Takeaway:



MVC pattern in MERN stack helps in building **well-structured**, **maintainable** and **scalable** applications by separating data, logic and UI into **Model**, **View** and **Controller**.



17.6 Controllers

17.6.1 What are Controllers?

- * Controllers handle incoming requests, process business logic and send responses to the client.
- * In MERN, controllers act as an intermediary between routes and models.
- * They keep the application logic organized and reusable.

17.6.2 Role of Controllers

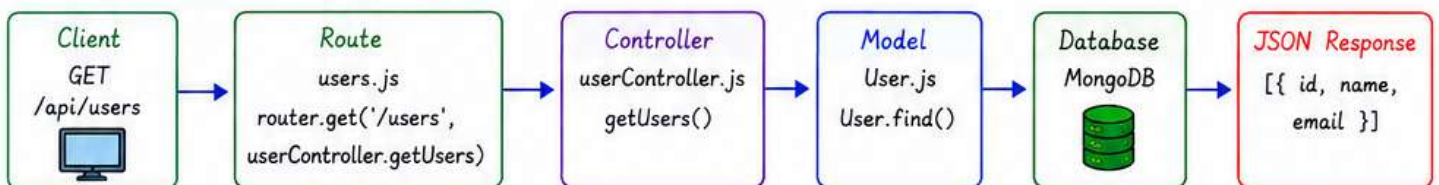
- * Receive HTTP requests from routes.
- * Validate input (if required).
- * Call the model to interact with the database.
- * Process the data and send appropriate response.
- * Handle errors and return meaningful messages.



17.6.4 Types of Controllers

1. **Single Resource Controller**
Handles CRUD operations for a single resource.
Example: UserController.js
2. **Resource Controller**
Handles multiple related resources.
Example: AuthController.js
3. **Custom Controller**
Contains specific business logic not directly tied to CRUD operations.

17.6.6 Example Flow: Get All Users



Key Points:

- ✓ Controllers connect routes and models.
- ✓ They contain the application logic.
- ✓ Use proper error handling and status codes.
- ✓ Well-structured controllers make the code scalable and maintainable.



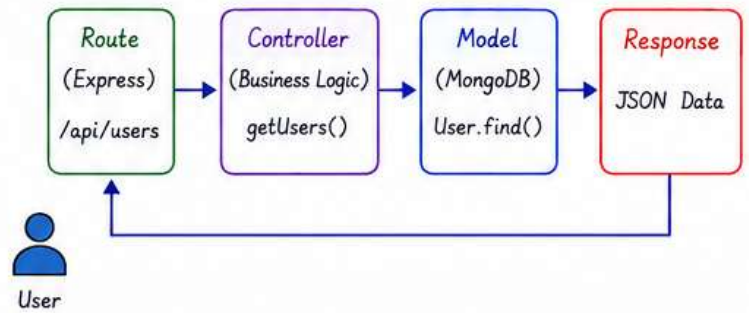
Key Takeaway:



Controllers are the **brain** of the application in MERN stack. They **process requests**, interact with models and send **meaningful responses** to the client, ensuring a **smooth** and **secure** flow of data.



Controller in MERN Flow



17.6.3 Example Controller (User Controller)

```
const User = require('../models/User');
exports.getUsers = async (req, res) => {
  try {
    const users = await User.find();
    res.status(200).json(users);
  } catch (err) {
    res.status(500).json({ message: 'Server Error' });
  }
};
```

17.6.5 Best Practices

- * Keep controllers thin—only handle business logic.
- * Use async/await for asynchronous operations.
- * Validate input before processing.
- * Send proper HTTP status codes.
- * Handle errors gracefully.
- * Keep controllers modular and organized.

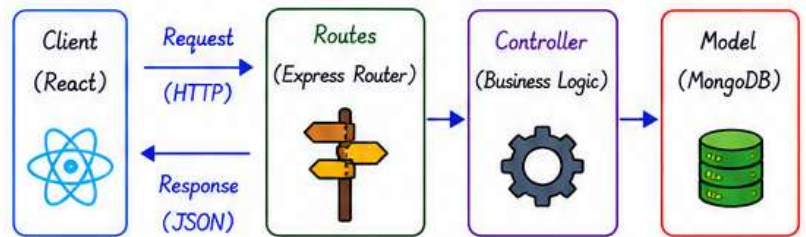


17.7 Routes

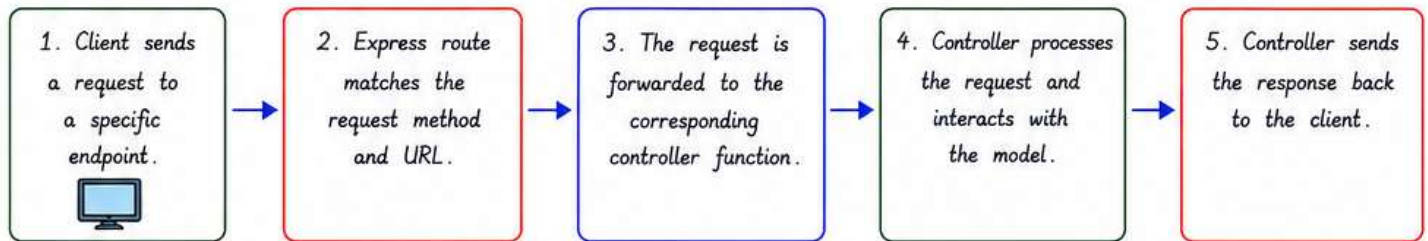
17.7.1 What are Routes?

- * Routes define the **endpoints** of an API.
- * They map HTTP methods (GET, POST, PUT, DELETE) to controller functions.
- * Routes act as a **bridge** between the client request and the controller.
- * In MERN (Express), routes are defined using **Express Router**.

Route in MERN Flow



17.7.2 How Routes Work?



Key Points:

- ✓ Routes define the API endpoints.
- ✓ Each route is associated with an HTTP method.
- ✓ Routes forward requests to controller functions.
- ✓ Routes help in organizing and managing API structure.
- ✓ They make the application modular and easy to maintain.



17.7.3 Example Routes (User Management)

Method	Endpoint	Description	Controller Function
GET	/api/users	Get all users	getUsers()
GET	/api/users/:id	Get user by ID	getUserById()
POST	/api/users	Create a new user	createUser()
PUT	/api/users/:id	Update user by ID	updateUser()
DELETE	/api/users/:id	Delete user by ID	deleteUser()

17.7.4 Example: User Routes (userRoutes.js)

```
const express = require('express');
const router = express.Router();
const userController = require('../controllers/userController');

// Get all users
router.get('/', userController.getUsers);

// Get user by ID
router.get('/:id', userController.getUserById);

// Create a new user
router.post('/', userController.createUser);

// Update user by ID
router.put('/:id', userController.updateUser);

// Delete user by ID
router.delete('/:id', userController.deleteUser);

module.exports = router;
```

Explanation:

- * We create a Router object.
- * Define routes for different HTTP methods.
- * Use route is linked to a controller function.
- * Export the router to use in app.js



17.7.5 Best Practices

- * Keep routes modular (separate files for each entity).
- * Use meaningful and consistent endpoint names.
- * Use proper HTTP methods for each operation.
- * Validate input data in controllers.
- * Handle errors and send proper status codes.
- * Keep route files clean and easy to understand.



Key Takeaway:



Routes define the **endpoints** of an API and map HTTP methods to controller functions. They play a crucial role in handling client requests and building a **well-structured, scalable** MERN application.

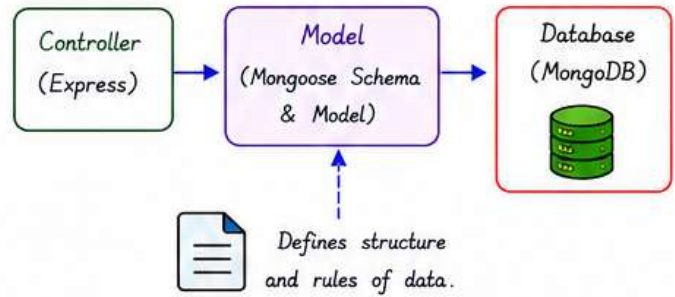


17.8 Models

17.8.1 What are Models?

- * Models represent the data structure and business logic of the application.
- * In MongoDB, models define the schema of the data.
- * In MERN, models are created using **Mongoose** (an ODM for **MongoDB**).
- * Models are used to interact with the database (CRUD operations).

Model in MERN Flow



17.8.2 Components of a Model

1. Schema

- * Defines the structure of the data.
- * Specifies fields, data types, validation rules, default values, etc.



2. Model

- * Created from a schema.
- * Provides an interface to the database for CRUD operations.



3. Documents

- * Actual data stored in a collection.
- * Follows the structure defined by the schema.



17.8.3 Example: User Model

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  age: { type: Number, default: 18 },
  createdAt: { type: Date, default: Date.now }
});

const User = mongoose.model('User', userSchema);
module.exports = User;
```

17.8.4 What is Happening?

- * We define a schema (userSchema) with fields and rules.
- * We create a model (User) from the schema.
- * The model is linked to the MongoDB collection (users by default).
- * We export the model to use in controllers.



Key Points:

- ✓ Models define the structure of the data.
- ✓ Mongoose models provide methods to perform CRUD operations.
- ✓ Models enforce data validation and rules.
- ✓ They make database interactions easy and organized.



17.8.5 Common Mongoose Model Methods

Method	Description	Example
<code>find()</code>	Find all documents	<code>User.find()</code>
<code>findById(id)</code>	Find a document by ID	<code>User.findById(id)</code>
<code>findOne(query)</code>	Find a single document	<code>User.findOne({ email: 'a@mail.com' })</code>
<code>findByIdAndUpdate(id, data)</code>	Update a document by ID	<code>User.findByIdAndUpdate(id, data)</code>
<code>findByIdAndDelete(id)</code>	Delete a document by ID	<code>User.findByIdAndDelete(id)</code>
<code>save()</code>	Save a new document	<code>new User(data).save()</code>

17.8.6 Example: Create and Save

```
const User = require('./models/User');
const createUser = async (req, res) => {
  try {
    const user = new User({
      name: 'Ravi',
      email: 'ravi@gmail.com',
      password: '123456',
      age: 22
    });
    await user.save();
    res.status(201).json({ message: 'User created', user });
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
};
```

17.8.7 Best Practices

- * Keep schemas clean and simple.
- * Use validation to maintain data integrity.
- * Use meaningful field names and data types.
- * Use indexes for fields that are frequently searched.
- * Handle errors properly in model operations.
- * Keep models and business logic separate.



Key Takeaway:



Models in MERN define the **structure** and **rules** of data using Mongoose. They provide **easy** methods to perform **CRUD** operations and ensure data **consistency**.

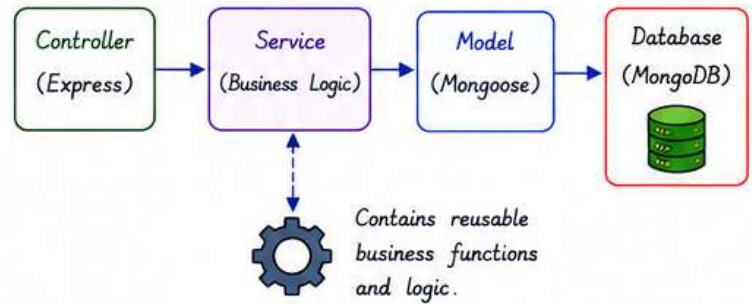


17.9 Services

17.9.1 What are Services?

- * Services contain the core **business logic** of the application.
- * They are used to keep controllers thin and maintain **separation of concerns**.
- * Services interact with models (database) and perform operations like validation, processing and computations.
- * In MERN, services are plain **JavaScript** files that export functions.

Service in MERN Flow



17.9.2 Why Use Services?

- ✓ Keeps controllers clean and focused.
- ✓ Improves code reusability.
- ✓ Easier to test and maintain.
- ✓ Centralizes business logic in one place.
- ✓ Makes the application scalable and modular.



17.9.3 Role of Services in MERN

- * Validate and sanitize data.
- * Perform business operations.
- * Communicate with models (database).
- * Return results to controllers.
- * Handle errors and exceptions.



17.9.4 Example: User Service Functions

```
const User = require('../models/User');

// Get all users
exports.getAllUsers = async () => {
  return await User.find();
};

// Get user by ID
exports.getUserById = async (id) => {
  return await User.findById(id);
};

// Create a new user
exports.createUser = async (userData) => {
  const user = new User(userData);
  return await user.save();
};

// Update user by ID
exports.updateUser = async (id, updateData) => {
  return await User.findByIdAndUpdate(id, updateData, { new: true });
};

// Delete user by ID
exports.deleteUser = async (id) => {
  return await User.findByIdAndDelete(id);
};
```

17.9.5 How Services are Used in Controller

```
const userService = require('../services/userService');

exports.getAllUsers = async (req, res) => {
  try {
    const users = await userService.getAllUsers();
    res.status(200).json(users);
  } catch (err) {
    res.status(500).json({ message: 'Server Error' });
  }
};
```

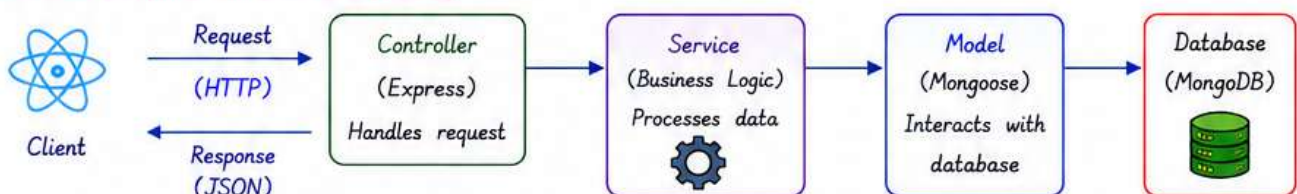


17.9.6 Best Practices

- * Keep services focused on business logic.
- * Do not handle HTTP requests/responses in services.
- * Use async/await for database operations.
- * Handle errors inside services and throw meaningful messages.
- * Keep services modular (one service per feature/entity).



17.9.7 Example Flow with Services



17.9.8 Advantages of Using Services



Better separation of concerns.



Code reusability in multiple places.



Easier maintenance and debugging.



Improves scalability of application.



Team collaboration becomes easier.

Key Takeaway:



Services are the heart of **business logic** in MERN applications.

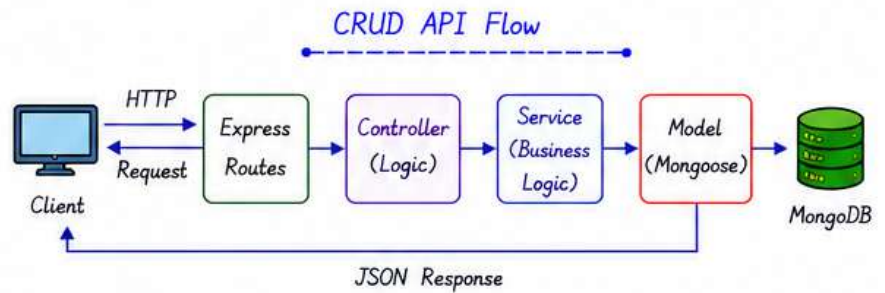
They help in keeping controllers **clean**, code **reusable**, and the application **scalable** and **maintainable**.



17.10 CRUD APIs

17.10.1 What are CRUD APIs?

- * CRUD stands for **Create, Read, Update, Delete**.
- * CRUD APIs perform basic operations on resources in a database.
- * They are the foundation of most web applications.
- * In MERN, CRUD APIs are built using Express (routes + controllers + models + services).

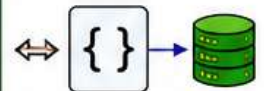


17.10.2 CRUD Operations

Operation	HTTP Method	Endpoint (Example: /api/users)	Description
Create	POST	/api/users	Create a new user
Read (All)	GET	/api/users	Get all users
Read (By ID)	GET	/api/users/:id	Get user by ID
Update (By ID)	PUT	/api/users/:id	Update user by ID
Delete (By ID)	DELETE	/api/users/:id	Delete user by ID

17.10.3 Example: User Resource

```
User {
  _id: ObjectId,
  name: String,
  email: String,
  password: String,
  age: Number,
  createdAt: Date
}
```



17.10.4 CRUD API Examples (User APIs)

1. CREATE - Create a new user (POST /api/users)

```
router.post('/', async (req, res) => {
  try {
    const user = new User(req.body);
    const savedUser = await user.save();
    res.status(201).json(savedUser);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});
```



2. READ - Get all users (GET /api/users)

```
router.get('/', async (req, res) => {
  try {
    const users = await User.find();
    res.status(200).json(users);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});
```



3. READ - Get user by ID (GET /api/users/:id)

```
router.get('/:id', async (req, res) => {
  try {
    const user = await User.findById(req.params.id);
    if (!user) return res.status(404).json({ message: 'User not found' });

    res.status(200).json(user);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});
```



4. UPDATE - Update user by ID (PUT /api/users/:id)

```
router.put('/:id', async (req, res) => {
  try {
    const updatedUser = await User.findByIdAndUpdate(
      req.params.id,
      req.body,
      { new: true, runValidators: true }
    );
    if (!updatedUser) return res.status(404).json({ message: 'User not found' });
    res.status(200).json(updatedUser);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});
```



5. DELETE - Delete user by ID (DELETE /api/users/:id)

```
router.delete('/:id', async (req, res) => {
  try {
    const deletedUser = await User.findByIdAndDelete(req.params.id);
    if (!deletedUser) return res.status(404).json({ message: 'User not found' });
    res.status(200).json({ message: 'User deleted successfully' });
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});
```



17.10.5 Sample Testing (Postman)

- * POST /api/users → Create user
- * GET /api/users → Get all users
- * GET /api/users/:id → Get user by ID
- * PUT /api/users/:id → Update user by ID
- * DELETE /api/users/:id → Delete user by ID



17.10.6 Example Request & Response (Create User)

Request (POST /api/users)

```
{
  "name": "Ravi",
  "email": "ravi@gmail.com",
  "password": "123456",
  "age": 22
}
```



Response (201 Created)

```
{
  "_id": "665f2d3b3f0a2c1b9a7c1234",
  "name": "Ravi",
  "email": "ravi@gmail.com",
  "age": 22,
  "createdAt": "2024-06-01T10:15:20.123Z",
  "__v": 0
}
```



17.10.7 Best Practices

- ✓ Use proper HTTP methods for each operation.
- ✓ Validate input data before processing.
- ✓ Handle errors and return meaningful messages.
- ✓ Use async/await for database operations.
- ✓ Use status codes correctly (201, 200, 404, 500).
- ✓ Keep routes, controllers, services and models separate.



Key Takeaway:



CRUD APIs are the backbone of any application. They allow **Create, Read, Update** and **Delete** operations on resources using standard HTTP methods in a structured way.

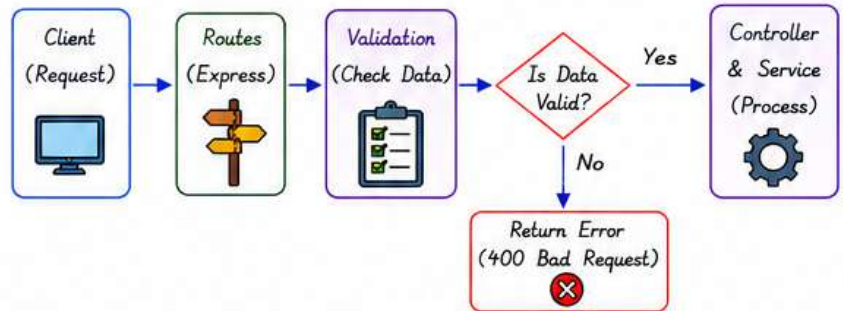


17.11 API Validation

17.11.1 What is API Validation?

- API Validation is the process of checking incoming data to ensure it is **correct**, **complete** and **safe**.
- It helps prevent invalid data from entering the database.
- Validation improves reliability, security and user experience.
- In MERN, validation is done in the backend (Express) before performing operations.

Validation Flow in API



17.11.2 Types of Validation

Type	Description
Required Validation	Checks if a field is provided.
Type Validation	Checks if the field has the correct data type.
Format Validation	Checks if the field follows a specific format (email, phone, date, etc.).
Length Validation	Checks minimum and maximum length.
Range Validation	Checks if the value is within a valid range.
Custom Validation	User-defined validation rules.

17.11.3 Example: Invalid Request

```
{
  "name": "",
  "email": "ravi#gmail.com",
  "age": "abc",
  "password": "123",
}
```

← Name is required
← Invalid email format
← Age should be a number
← Password too short

Response (400 Bad Request)

17.11.4 Validation using Joi (Example)

```
const Joi = require('joi');
// Joi schema for user validation
const userSchema = Joi.object({
  name: Joi.string().min(3).max(50).required(),
  email: Joi.string().email().required(),
  age: Joi.number().integer().min(18).max(100).required(),
  password: Joi.string().min(6).required()
});

// Middleware for validation
exports.validateUser = (req, res, next) => {
  const { error } = userSchema.validate(req.body);
  if (error) {
    return res.status(400).json({
      message: 'Validation Error',
      details: error.details.map(err => err.message)
    });
  }
  next(); // Data is valid, proceed to controller
};
```

17.11.5 Using Validation Middleware in Route

```
const express = require('express');
const router = express.Router();
const { validateUser } = require('../middleware/validation');
const userController = require('../controllers/userController');

// Apply validation before creating user
router.post('/', validateUser, userController.createUser);

module.exports = router;
```

17.11.6 Example: Validation Error Response

```
{
  "message": "Validation Error",
  "details": [
    "name is required",
    "email must be a valid email",
    "age must be a number",
    "password must be at least 6 characters long"
  ]
}
```

HTTP Status Code: 400

17.11.7 Manual Validation (Without Joi)

```
exports.validateUserManual = (req, res, next) => {
  const { name, email, age, password } = req.body;
  const errors = [];
  if (!name || name.trim() === '') errors.push('Name is required');
  if (!email || !email.includes('@')) errors.push('Invalid email');
  if (typeof age !== 'number') errors.push('Age must be a number');
  if (!password || password.length < 6) errors.push('Password too short');
  if (errors.length > 0) {
    return res.status(400).json({ message: 'Validation Error', errors });
  }
  next();
};
```

17.11.8 Best Practices

- Always validate data before processing.
- Validate both required fields and data types.
- Use libraries like Joi or express-validator for clean code.
- Return meaningful error messages.
- Use appropriate HTTP status codes (400 for validation errors).
- Sanitize and validate user input to prevent attacks.
- Keep validation logic in middleware or separate files.

17.11.9 Common Validation Rules (Joi)

Rule	Description	Rule	Description
required()	Field is mandatory.	max(length)	Maximum length or value.
string()	Field must be a string.	pattern(regex)	Field must match the regex pattern.
number()	Field must be a number.	integer()	Field must be an integer.
email()	Field must be a valid email.	valid(value1, value2)	Field must be one of the specified values.
min(length)	Minimum length or value.	custom()	Custom validation rule.

Key Takeaway:



API Validation ensures that only **correct**, **safe** and **meaningful** data is processed. It protects the application from bad data, improves security and keeps the database clean.

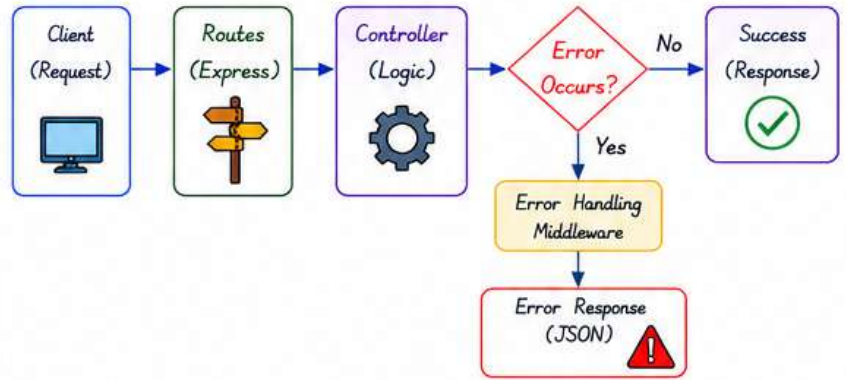


17.12 API Error Handling

17.12.1 What is API Error Handling?

- API Error Handling is the process of detecting, handling and responding to errors in a **controlled manner**.
- It ensures that the API does not crash and provides **meaningful feedback** to the client.
- Good error handling improves **reliability**, **debugging** and **user experience**.
- In MERN (Express), errors are handled using **try...catch**, custom error classes and middleware.

Error Handling Flow in API



17.12.2 Common Errors in APIs

Error Type	Description	Example
Validation Error	Invalid or missing input data.	Invalid email format
Resource Not Found	Requested resource not found.	User with given ID not found
Unauthorized Error	Authentication required.	Access token missing/invalid
Forbidden Error	No permission to access resource.	Admin access required
Server Error	Unexpected error on server.	Database connection failed
Bad Request	Invalid request format.	Malformed JSON

17.12.3 Standard Error Response Format

```
{
  "success": false,
  "message": "Human readable error message",
  "error": "Detailed error (optional)",
  "statusCode": 400,
  "data": null,
  "timestamp": "2024-06-01T10:15:20.123Z"
}
```

Always return consistent error responses.



17.12.4 Example: Handling Errors in Controller

```
const User = require('../models/User');

// Get user by ID
exports.getUserById = async (req, res, next) => {
  try {
    const user = await User.findById(req.params.id);
    if (!user) {
      const err = new Error('User not found');
      err.statusCode = 404;
      return next(err); // Pass error to middleware
    }
    res.status(200).json({ success: true, data: user });
  } catch (err) {
    next(err); // Pass any other error
  }
};
```



17.12.5 Global Error Handling Middleware

```
// errorMiddleware.js
const errorMiddleware = (err, req, res, next) => {
  const statusCode = err.statusCode || 500;
  const message = err.message || 'Internal Server Error';

  console.error(err.stack); // Log error

  res.status(statusCode).json({
    success: false,
    message: message,
    error: process.env.NODE_ENV === 'production'
      ? undefined : err.stack,
    statusCode: statusCode,
    data: null,
    timestamp: new Date().toISOString()
  });
};
module.exports = errorMiddleware;
```



17.12.6 Using Error Middleware in App

```
const express = require('express');
const app = express();

// ... other middlewares and routes
const userRoutes = require('../routes/userRoutes');
app.use('/api/users', userRoutes);

// Global Error Handling Middleware
const errorMiddleware = require('../middleware/errorMiddleware');
app.use(errorMiddleware); // Must be last

module.exports = app;
```



17.12.7 Example Error Responses

Error	Status Code	Example Response (JSON)
Validation Error	400 Bad Request	{ "success": false, "message": "Invalid email", "statusCode": 400 }
Not Found	404 Not Found	{ "success": false, "message": "User not found", "statusCode": 404 }
Unauthorized	401 Unauthorized	{ "success": false, "message": "Unauthorized access", "statusCode": 401 }
Forbidden	403 Forbidden	{ "success": false, "message": "Access denied", "statusCode": 403 }
Server Error	500 Internal Server Error	{ "success": false, "message": "Internal Server Error", "statusCode": 500 }

17.12.9 Common HTTP Status Codes

Status Code	Meaning	When to Use
200 OK	Request successful	Data retrieved/created/updated/deleted
400 Bad Request	Invalid request	Validation failed, wrong format
401 Unauthorized	Authentication required	Token missing or invalid
403 Forbidden	Access denied	No permission to access resource
404 Not Found	Resource not found	Requested data does not exist
500 Internal Server Error	Server error	Unexpected server-side error

17.12.8 Best Practices

- Always validate input data before processing.
- Use meaningful and user-friendly error messages.
- Return appropriate HTTP status codes.
- Do not expose sensitive server errors in production.
- Log errors on the server for debugging.
- Use a centralized error handling middleware.
- Keep error responses consistent across all APIs.



Key Takeaway:



Proper API Error Handling makes your application **reliable**, **secure** and **user-friendly**. Always handle errors gracefully and provide clear feedback to the client.

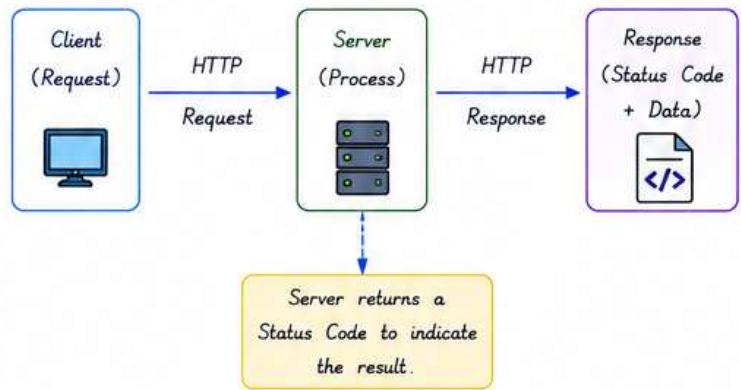


17.13 HTTP Status Codes

17.13.1 What are HTTP Status Codes?

- HTTP Status Codes are **3-digit numbers** returned by the server in response to a client request.
- They indicate whether the request was **successful**, **redirected**, an **error** occurred, etc.
- Status codes help clients understand the result of their request.
- In APIs, status codes are always returned along with a response body.

Request - Response Flow



17.13.2 Categories of HTTP Status Codes

Category	Range	Description	Example Code
1xx Informational	100 - 199	Request received, continuing process	100 Continue
2xx Success	200 - 299	Request successfully received, understood and accepted	200 OK
3xx Redirection	300 - 399	Further action needs to be taken by the client	301 Moved Permanently
4xx Client Error	400 - 499	Request contains bad syntax or cannot be fulfilled	400 Bad Request
5xx Server Error	500 - 599	Server failed to fulfill a valid request	500 Internal Server Error

17.13.3 Most Common HTTP Status Codes

Status Code	Meaning
200 OK	Request successful.
201 Created	Resource created successfully.
204 No Content	Request successful, no content to return.
301 Moved Permanently	Resource moved permanently to a new URL.
400 Bad Request	Invalid request due to client error.
401 Unauthorized	Authentication is required.
403 Forbidden	Server understood request but refuses to authorize it.
404 Not Found	Requested resource not found.
409 Conflict	Conflict with the current state of resource.
500 Internal Server Error	Unexpected server error.
503 Service Unavailable	Server is currently unavailable.

17.13.4 Example: API Responses with Status Codes

- 200 OK - Request successful

```
{
  "success": true,
  "message": "User retrieved successfully",
  "data": { "id": 1, "name": "Ravi" }
}
```
- 201 Created - Resource created

```
{
  "success": true,
  "message": "User created successfully",
  "data": { "id": 2, "name": "Ravi" }
}
```
- 400 Bad Request - Invalid input

```
{
  "success": false,
  "message": "Invalid email format",
  "errors": ["email must be valid"]
}
```
- 404 Not Found - Resource not found

```
{
  "success": false,
  "message": "User not found",
  "statusCode": 404
}
```
- 500 Internal Server Error - Server error

```
{
  "success": false,
  "message": "Something went wrong",
  "statusCode": 500
}
```
- 503 Service Unavailable - Server busy

```
{
  "success": false,
  "message": "Service temporarily unavailable",
  "statusCode": 503
}
```

17.13.5 Where Status Codes are Used?

- To inform the client about the result of the request.
- To handle errors and exceptions properly.
- To show meaningful messages to users.
- To maintain standard communication between client and server.
- Helps in debugging and monitoring APIs.



17.13.6 Best Practices

- Always return appropriate status codes.
- Use 2xx for success, 4xx for client errors, 5xx for server errors.
- Provide meaningful messages in response.
- Do not expose sensitive server information.
- Be consistent in all API endpoints.



17.13.7 Detailed List of HTTP Status Codes

Category	Status Code	Meaning	When to Use	Category	Status Code	Meaning	When to Use
1xx Informational	100 Continue	Request received, continue.	Client should continue request.	4xx Client Error	400 Bad Request	Invalid request.	Validation error, wrong data.
	101 Switching Protocols	Switching protocols.	Protocol change.		401 Unauthorized	Authentication required.	No/invalid token.
2xx Success	200 OK	Request successful.	GET/PUT/DELETE success.		403 Forbidden	Access denied.	No permission.
	201 Created	Resource created.	POST success.		404 Not Found	Resource not found.	Invalid endpoint or ID.
	202 Accepted	Request accepted.	Processing in background.		409 Conflict	Conflict with resource state.	Duplicate or conflict data.
	204 No Content	Success, no content to return.	DELETE success.		429 Too Many Requests	Too many requests.	Rate limiting.
3xx Redirection	301 Moved Permanently	Resource moved permanently.	URL changed permanently.	5xx Server Error	500 Internal Server Error	Server error.	Unexpected server failure.
	302 Found	Resource found temporarily.	Temporary redirect.		502 Bad Gateway	Invalid response from server.	Gateway/proxy error.
	304 Not Modified	Resource not modified.	Use cached data.		503 Service Unavailable	Server unavailable.	Server down/maintenance.
					504 Gateway Timeout	Server timeout.	Upstream server timeout.

Key Takeaway:



HTTP Status Codes are the **language of APIs**.

They help in **clear communication**, error handling and building reliable applications.

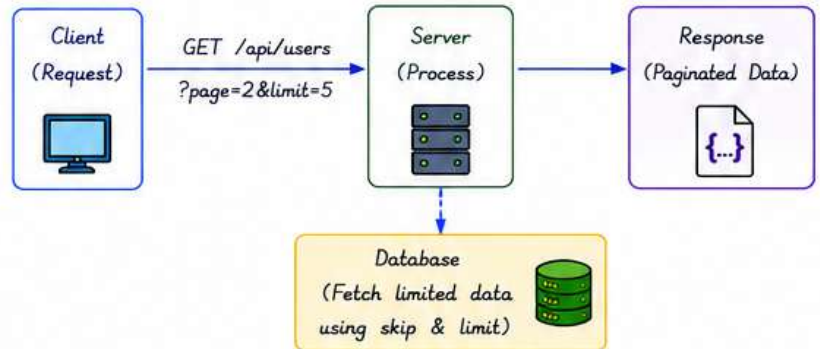


17.14 API Pagination

17.14.1 What is API Pagination?

- Pagination is the process of dividing a large set of data into **smaller chunks (pages)**.
- It helps improve **performance**, reduce load on the server and enhance **user experience**.
- Instead of returning all records at once, we return a specific page of results.
- Common query parameters: **page**, **limit**, **skip**, **offset**, **nextPage**, **totalPages**, etc.
- Used in list APIs like users, products, orders, posts, etc.

Pagination Flow in API



17.14.2 Pagination Parameters

Parameter	Description	Default	Example
page	Page number requested	1	page=2
limit	Number of items per page	10	limit=5
skip / offset	Number of items to skip	0	skip=10
sort	Field to sort by	_id	sort=name:asc
order	Sort order (asc / desc)	asc	order=desc
search	Search keyword	-	search=ravi

17.14.3 Example Request

GET /api/users?page=2&limit=5&sort=name&order=asc

Meaning:

- page=2 → Return page number 2
- limit=5 → 5 items per page
- sort=name → Sort by name
- order=asc → Ascending order



17.14.4 Example Response

```
{
  "success": true,
  "data": [
    { "_id": "101", "name": "Amit", "email": "amit@gmail.com" },
    { "_id": "102", "name": "Neha", "email": "neha@gmail.com" },
    { "_id": "103", "name": "Ravi", "email": "ravi@gmail.com" },
    { "_id": "104", "name": "Pooja", "email": "pooja@gmail.com" },
    { "_id": "105", "name": "Kiran", "email": "kiran@gmail.com" }
  ],
  "pagination": {
    "currentPage": 2,
    "totalPages": 4,
    "totalItems": 20,
    "itemsPerPage": 5,
    "hasNextPage": true,
    "hasPrevPage": true,
    "nextPage": 3,
    "prevPage": 1
  }
}
```

Only 5 items
(returned)
for page = 2

17.14.6 Best Practices

- Always use pagination for large datasets.
- Set a default limit and maximum limit (e.g., max 100).
- Validate page and limit values.
- Use proper sorting for consistent results.
- Return useful pagination metadata (totalPages, hasNextPage, etc.).
- Use indexes on fields used for sorting and searching.
- Avoid returning unnecessary fields.



17.14.7 Example Request & Response (Summary)

Request

GET /api/users?page=3&limit=4&sort=name&order=desc

Query Params

- page = 3
- limit = 4
- sort = name
- order = desc

Response (Summary)

- Data: 4 users
- currentPage: 3
- totalPages: 5
- totalItems: 20
- hasNextPage: true
- hasPrevPage: true



17.14.8 How Pagination Works (Calculation)

Page	Limit (Items per page)	Skip (Offset)	Items Returned	Explanation
1	5	(1-1) * 5 = 0	1 - 5	First 5 items
2	5	(2-1) * 5 = 5	6 - 10	Next 5 items
3	5	(3-1) * 5 = 10	11 - 15	Next 5 items
n	5	(n-1) * 5	...	...

Formula

skip = (page - 1) * limit
totalPages = ceil(totalItems / limit)
hasNextPage = page < totalPages



Key Takeaway:



Pagination makes APIs **fast**, **efficient** and **scalable**.

It improves performance and user experience by returning only the required data.



17.14.5 Pagination in Controller (Example)

```
exports.getAllUsers = async (req, res) => {
  try {
    // Get query parameters
    const page = parseInt(req.query.page) || 1;
    const limit = parseInt(req.query.limit) || 10;
    const skip = (page - 1) * limit;
    const sortBy = req.query.sort || '_id';
    const order = req.query.order || 'asc';
    const search = req.query.search || '';

    // Build filter
    let filter = {};
    if (search) {
      filter.name = { $regex: search, $options: 'i' };
    }

    // Query with pagination
    const users = await User.find(filter)
      .sort({ [sortBy]: order === 'desc' ? -1 : 1 })
      .skip(skip)
      .limit(limit);

    const totalItems = await User.countDocuments(filter);
    const totalPages = Math.ceil(totalItems / limit);

    res.status(200).json({
      success: true,
      data: users,
      pagination: {
        currentPage: page,
        totalPages,
        totalItems,
        itemsPerPage: limit,
        hasNextPage: page < totalPages,
        hasPrevPage: page > 1,
        nextPage: page < totalPages ? page + 1 : null,
        prevPage: page > 1 ? page - 1 : null
      }
    });
  } catch (err) {
    res.status(500).json({ success: false, message: err.message });
  }
};
```

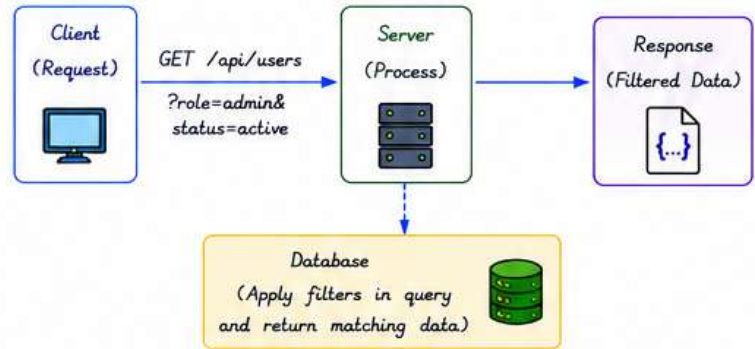


17.15 API Filtering

Filtering Flow in API

17.15.1 What is API Filtering?

- Filtering allows clients to retrieve only the data that matches **specific conditions**.
- It helps in **narrowing down** large datasets based on one or more criteria.
- Improves **performance**, reduces bandwidth usage and enhances **user experience**.
- Commonly used in list APIs like users, products, orders, posts, etc.



17.15.2 Common Filter Parameters

Parameter	Description	Example	Example URL
field=value	Equals match	role=admin	/api/users?role=admin
field_ne=value	Not equals	status_ne=inactive	/api/users?status_ne=inactive
field_gt=value	Greater than	age_gt=25	/api/users?age_gt=25
field_lt=value	Less than	age_lt=30	/api/users?age_lt=30
field_gte=value	Greater than or equal	age_gte=18	/api/users?age_gte=18
field_lte=value	Less than or equal	age_lte=60	/api/users?age_lte=60
field_in=a,b,c	In a list of values	role_in=admin,user	/api/users?role_in=admin,user
search=keyword	Keyword search	search=ravi	/api/users?search=ravi

17.15.3 Example Request

GET /api/users?role=admin&status=active&age_gt=20&sort=-createdAt

Meaning:

- role=admin → Only admin users
- status=active → Only active users
- age_gt=20 → Age greater than 20
- sort=-createdAt → Sort by createdAt in descending order



17.15.4 Example Response

```
{
  "success": true,
  "count": 3,
  "data": [
    { "_id": "101", "name": "Ravi", "role": "admin",
      "status": "active", "age": 28, "createdAt": "2024-06-01" },
    { "_id": "105", "name": "Amit", "role": "admin",
      "status": "active", "age": 32, "createdAt": "2024-05-20" },
    { "_id": "110", "name": "Neha", "role": "admin",
      "status": "active", "age": 25, "createdAt": "2024-05-10" }
  ],
  "totalPages": 1,
  "currentPage": 1
}
```

Only matching records are returned.

17.15.6 Best Practices

- Allow filtering only on specific fields to prevent misuse.
- Validate and sanitize query parameters.
- Combine filtering with pagination for better performance.
- Use indexes on database fields used for filtering.
- Set a default limit to avoid fetching huge datasets.
- Return meaningful messages for invalid filters.
- Avoid exposing internal implementation details in errors.



17.15.5 Filtering in Controller (Example)

```
exports.getUsers = async (req, res) => {
  try {
    let filter = {};
    // Apply filters based on query parameters
    if (req.query.role) filter.role = req.query.role;
    if (req.query.status) filter.status = req.query.status;
    if (req.query.age_gt) filter.age = { $gt: parseInt(req.query.age_gt) };
    if (req.query.age_lt) filter.age = { $lt: parseInt(req.query.age_lt) };
    if (req.query.age_gte) filter.age = { $gte: parseInt(req.query.age_gte) };
    if (req.query.age_lte) filter.age = { $lte: parseInt(req.query.age_lte) };
    if (req.query.role_in) filter.role = { $in: req.query.role_in.split(',') };

    // Keyword search (example on name & email)
    if (req.query.search) {
      filter.$or = [
        { name: { $regex: req.query.search, $options: 'i' } },
        { email: { $regex: req.query.search, $options: 'i' } }
      ];
    }

    // Sorting
    let sort = req.query.sort || '-createdAt';

    const users = await User.find(filter).sort(sort);
    res.status(200).json({ success: true, count: users.length, data: users });
  } catch (err) {
    res.status(500).json({ success: false, message: err.message });
  }
};
```



17.15.8 Example: Request & Response Summary

Request

GET /api/users?
role_in=admin,user&
status=active&age_gte=18&
age_lte=40&sort=name&limit=5&
page=1



Response (Summary)

- success: true
- count: 5
- data: [... matching users ...]
- currentPage: 1
- totalPages: 2



17.15.7 Common Filters with Examples

Filter Type	Syntax	Example	Description
Equals	field=value	role=admin	Matches exact value
Not Equals	field_ne=value	role_ne=user	Does not match value
Greater Than	field_gt=value	age_gt=20	Value greater than
Less Than	field_lt=value	age_lt=20	Value less than
In List	field_in=a,b	status_in=active,inactive	Matches any in list
Keyword Search	search=keyword	search=ram	Search in text fields

17.15.9 Example: Filters on Different Fields

Field	Possible Filters	Example	Example URL	Explanation
name	equals, search	name=Ravi / search=Ravi	/api/users?name=Ravi /api/users?search=Ravi	Get user with exact name Search users by name
role	equals, in	role=admin / role_in=admin,user	/api/users?role=admin /api/users?role_in=admin,user	Get admin users Get admin or user roles
status	equals, ne, in	status=active / status_ne=inactive status_in=active,inactive	/api/users?status=active /api/users?status_ne=inactive /api/users?status_in=active,inactive	Active users Exclude inactive Active OR inactive
age	gt, lt, gte, lte	age_gt=20 / age_lte=30	/api/users?age_gt=20 /api/users?age_lte=30	Age greater than 20 Age less than or equal to 30
createdAt	gt, lt, gte, lte	createdAt_gte=2024-01-01 createdAt_lt=2024-06-01	/api/users?createdAt_gte=2024-01-01 /api/users?createdAt_lt=2024-06-01	Created on or after date Created before date

Key Takeaway:



API Filtering helps return only the **data you need**.

It reduces load, improves **performance** and enhances the overall **user experience**.

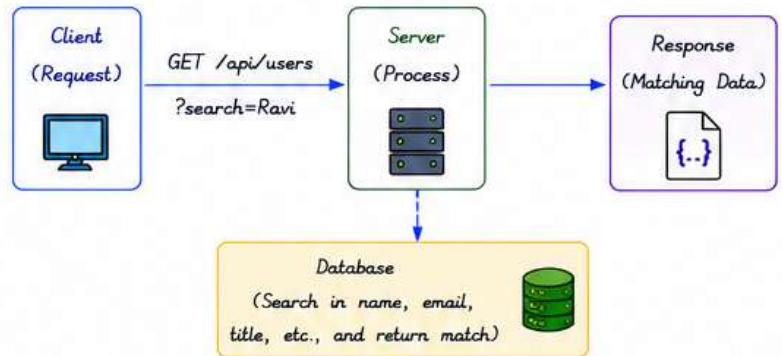


17.16 API Searching

Searching Flow in API

17.16.1 What is API Searching?

- Searching allows clients to find data that matches a **text query**.
- It is used to search by **keyword**, **name**, **email**, **title**, etc.
- Improves **user experience** by quickly finding relevant results.
- Commonly used in list APIs like users, products, orders, posts, etc.



17.16.2 Common Search Parameters

Parameter	Description	Example	Example URL
search / q	Text to search in records	search=Ravi	/api/users?search=Ravi
searchFields	Fields to apply search on	searchFields=name,email	/api/users?search=Ravi&searchFields=name,email
matchType	Type of match (contains, startsWith, exact, etc.)	matchType=contains	/api/users?search=Ravi&matchType=contains
case	Case sensitivity (insensitive / sensitive)	case=insensitive	/api/users?search=Ravi&case=insensitive
minLength	Minimum characters to start searching	minLength=2	/api/users?search=Ra&minLength=2
page, limit	Pagination with search	page=1&limit=5	/api/users?search=Ravi&page=1&limit=5

17.16.3 Example Request

GET /api/users?search=Ravi&searchFields=name,email&matchType=contains&case=insensitive&page=1&limit=5

Meaning:

- search=Ravi → Search text is "Ravi"
- searchFields=name,email → Search in name and email
- matchType=contains → Match if text contains "Ravi"
- case=insensitive → Case insensitive search
- page=1, limit=5 → Pagination: first page, 5 items



17.16.4 Example Response

```
{
  "success": true,
  "count": 2,
  "data": [
    { "_id": "101", "name": "Ravi Kumar", "email": "ravi@gmail.com" },
    { "_id": "115", "name": "Ravindra Singh", "email": "ravi.singh@gmail.com" }
  ],
  "pagination": {
    "currentPage": 1,
    "totalPages": 1,
    "totalItems": 2,
    "itemsPerPage": 5,
    "hasNextPage": false,
    "hasPrevPage": false
  }
}
```

Only matching records are returned.

17.16.6 Best Practices

- Provide simple and consistent search parameters.
- Allow searching on specific fields only.
- Use proper indexing on searchable fields.
- Implement minimum character length for search (e.g., minLength=2).
- Return paginated results with search.
- Validate and sanitize user input.
- Handle empty search and no-result cases gracefully.
- Do not expose sensitive fields in results.



17.16.7 Common Search Types with Examples

Search Type	Description	Example Query
Contains	Text contains the given value	search=Ravi (matches "Ravi Kumar")
Starts With	Text starts with the value	matchType=startsWith&search=Ra (matches "Ravi", "Raman")
Exact Match	Text exactly matches	matchType=exact&search=Ravi Kumar
Case Insensitive	Ignores case	case=insensitive&search=RAVI
Multiple Fields	Search in multiple fields	searchFields=name,email&search=Ravi

17.16.8 Example: Search Summary

Request
GET /api/users?search=Ravi&searchFields=name,email&matchType=contains&case=insensitive&page=1&limit=5

Response (Summary)
• count: 2
• data: [... matching users ...]
• pagination: { currentPage: 1, totalPages: 1, totalItems: 2 }

Key Takeaway:



API Searching helps users find the exact data they need quickly. It improves **usability**, saves bandwidth and enhances overall **user experience**.



17.16.5 Searching in Controller (Example)

```
exports.getUsers = async (req, res) => {
  try {
    let { search = '', searchFields = 'name,email',
        matchType = 'contains', case: caseType = 'insensitive',
        page = 1, limit = 10 } = req.query;

    // Build search filter
    let filter = {};
    if (search) {
      let fields = searchFields.split(',');
      let conditions = fields.map(field => {
        if (matchType === 'startsWith')
          return { [field]: { $regex: '^' + search, $options: caseType === 'insensitive' ? 'i' : '' } };
        else if (matchType === 'exact')
          return { [field]: { $regex: '^' + search + '$', $options: caseType === 'insensitive' ? 'i' : '' } };
        else // contains (default)
          return { [field]: { $regex: search, $options: caseType === 'insensitive' ? 'i' : '' } };
      });
      filter.$or = conditions;
    }

    // Pagination
    let skip = (parseInt(page) - 1) * parseInt(limit);
    let users = await User.find(filter).skip(skip).limit(parseInt(limit));
    let totalItems = await User.countDocuments(filter);
    let totalPages = Math.ceil(totalItems / parseInt(limit));

    res.status(200).json({
      success: true,
      count: users.length,
      data: users,
      pagination: {
        currentPage: parseInt(page), totalPages, totalItems,
        itemsPerPage: parseInt(limit),
        hasNextPage: page < totalPages, hasPrevPage: page > 1
      }
    });
  } catch (err) {
    res.status(500).json({ success: false, message: err.message });
  }
};
```

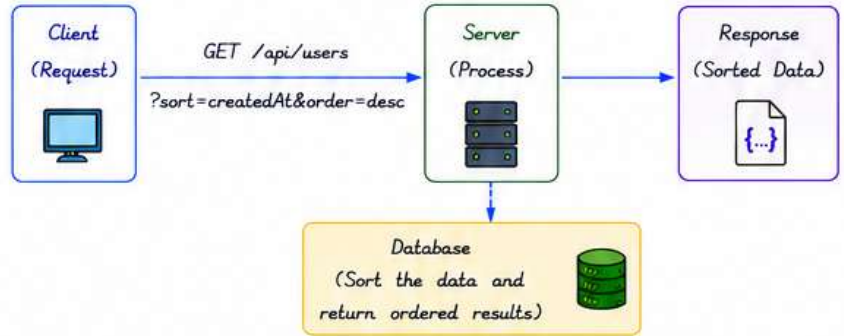


17.17 API Sorting

Sorting Flow in API

17.17.1 What is API Sorting?

- Sorting arranges the data in a specific order based on one or more fields.
- It helps clients to view data in **ascending (asc)** or **descending (desc)** order.
- Improves readability of data and enhances **user experience**.
- Commonly used in list APIs like users, products, orders, etc.



17.17.2 Common Sorting Parameters

Parameter	Description	Example	Example URL
sort	Field to sort by	createdAt, name, email, age	/api/users?sort=name
order	Sort order (asc / desc)	asc, desc	/api/users?order=desc
sortBy	Alias for sort	name	/api/users?sortBy=name
direction	Alias for order	asc, desc	/api/users?direction=asc
page, limit	Used with sorting (for pagination)	page=1&limit=10	/api/users?sort=name&order=asc&page=1&limit=10

17.17.3 Example Request

GET /api/users?sort=createdAt&order=desc&page=1&limit=5

Meaning:

- sort=createdAt → Sort by createdAt field
- order=desc → Descending order (latest first)
- page=1 → First page
- limit=5 → 5 items per page



17.17.4 Example Response

```
{
  "success": true,
  "count": 3,
  "data": [
    { "_id": "110", "name": "Neha", "createdAt": "2024-06-05T10:30:00Z" },
    { "_id": "109", "name": "Amit", "createdAt": "2024-06-04T09:15:00Z" },
    { "_id": "108", "name": "Ravi", "createdAt": "2024-06-03T08:20:00Z" }
  ],
  "pagination": {
    "currentPage": 1,
    "totalPages": 2,
    "totalItems": 10,
    "itemsPerPage": 5,
    "hasNextPage": true,
    "hasPrevPage": false
  }
}
```

Data is sorted by createdAt in descending order.

17.17.6 Best Practices

- Always allow sorting only on valid, indexed fields.
- Set a default sort field and order.
- Validate sort and order parameters.
- Use pagination with sorting for large datasets.
- Restrict sorting on sensitive fields.
- Provide clear API documentation.
- Return consistent results for same query.



17.17.7 Common Sort Fields with Examples

Field	Example	Description
name	?sort=name&order=asc	Sort by user name (A → Z)
email	?sort=email&order=asc	Sort by email (A → Z)
createdAt	?sort=createdAt&order=desc	Sort by creation date (latest first)
age	?sort=age&order=asc	Sort by age (smallest first)
price	?sort=price&order=desc	Sort by price (high to low)

17.17.8 Example: Requests & Responses

Request	Response (Summary)
GET /api/users?sort=name&order=asc	- Sorted by name (A → Z) - First page of results - Pagination info included
GET /api/users?sort=age&order=desc&page=2&limit=5	- Sorted by age (high to low) - Page 2, 5 items per page - Pagination info included
GET /api/users?sort=createdAt&order=asc	- Sorted by oldest first - All other pagination defaults

17.17.9 Common Order Values

Order Value	Meaning
asc	Ascending order (A → Z, 0 → 9, oldest → latest)
desc	Descending order (Z → A, 9 → 0, latest → oldest)

Note:

- Default order is usually 'asc'
- Always specify order with sort for clarity.



Key Takeaway:



API Sorting helps organize data in meaningful order. It improves **usability**, **performance** and delivers a better **user experience**.



```
exports.getUsers = async (req, res) => {
  try {
    // Get query parameters
    const sortField = req.query.sort || 'createdAt';
    const order = req.query.order || 'asc';

    // Build sort object
    const sortOrder = order === 'desc' ? -1 : 1;
    const sortOption = { [sortField]: sortOrder };

    // Pagination
    const page = parseInt(req.query.page) || 1;
    const limit = parseInt(req.query.limit) || 10;
    const skip = (page - 1) * limit;

    // Fetch sorted data
    const users = await User.find({})
      .sort(sortOption)
      .skip(skip)
      .limit(limit);

    const totalItems = await User.countDocuments({});
    const totalPages = Math.ceil(totalItems / limit);

    res.status(200).json({
      success: true,
      count: users.length,
      data: users,
      pagination: {
        currentPage: page,
        totalPages,
        totalItems,
        itemsPerPage: limit,
        hasNextPage: page < totalPages,
        hasPrevPage: page > 1
      }
    });
  } catch (err) {
    res.status(500).json({ success: false, message: err.message });
  }
};
```

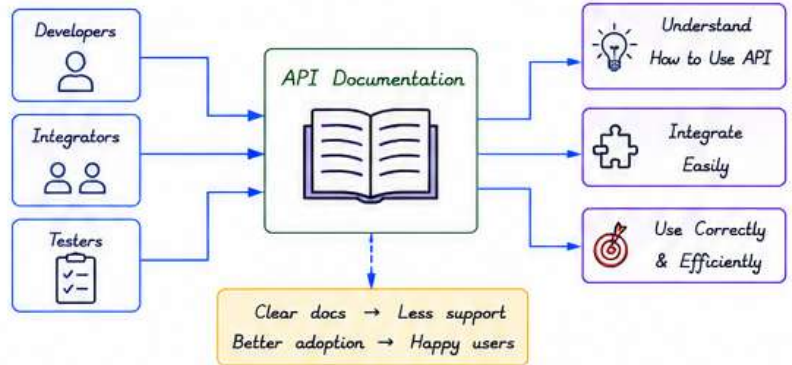


17.18 API Documentation

Role of API Documentation

17.18.1 What is API Documentation?

- * API Documentation is a detailed guide that explains how to **use an API**.
- * It helps developers understand available **endpoints**, **request parameters**, **responses**, **errors**, **authentication**, and **examples**.
- * Good documentation improves **developer experience** and adoption.
- * Commonly created using tools like **Swagger (OpenAPI)**, **Postman**, **ReadMe**, or **MkDocs**.



17.18.2 Key Sections in API Documentation

Section	Description
Overview	Brief introduction about the API and its purpose.
Base URL	The base URL of the API.
Authentication	How to authenticate requests (API Key, Token, etc.).
Endpoints	List of available API endpoints.
Request Parameters	Details of query, path, header and body parameters.
Request Example	Sample request with URL, headers, body.
Response Format	Description of successful response structure.
Error Codes	List of possible errors and their meanings.
Examples	Real-world request and response examples.
Rate Limits	Information about usage limits.
Contact / Support	How to get help or report issues.

17.18.3 Example: Endpoint Documentation

Endpoint : GET /api/users
Description : Get a list of users
Base URL : https://api.example.com
Method : GET
Authentication: Bearer Token (JWT)
Rate Limit : 100 requests per minute



Query Parameters

Parameter	Type	Required	Description	Example
page	integer	No	Page number	1
limit	integer	No	Items per page	10
sort	string	No	Sort by field	name
order	string	No	Sort order (asc/desc)	asc

Request Example

GET /api/users?page=1&limit=10&sort=name&order=asc
Authorization: Bearer <token>

Response Example (200 OK)

```
{
  "success": true,
  "data": [
    { "id": 1, "name": "Ravi", "email": "ravi@gmail.com" },
    { "id": 2, "name": "Neha", "email": "neha@gmail.com" }
  ],
  "pagination": {
    "currentPage": 1,
    "totalPages": 5,
    "totalItems": 48,
    "itemsPerPage": 10
  }
}
```

Error Responses

Status Code	Error	Description	Example
400	Bad Request	Invalid parameters	{ "message": "Invalid page" }
401	Unauthorized	Missing/invalid token	{ "message": "Unauthorized" }
404	Not Found	Resource not found	{ "message": "User not found" }
500	Server Error	Internal server error	{ "message": "Something went wrong" }

17.18.4 Example: Create User (POST)

Endpoint : POST /api/users
Description : Create a new user
Headers : Content-Type: application/json
Request Body:

```
{
  "name": "Ravi Kumar",
  "email": "ravi@gmail.com",
  "password": "password123"
}
```

Response (201 Created)

```
{
  "success": true,
  "message": "User created",
  "data": {
    "id": 101,
    "name": "Ravi Kumar",
    "email": "ravi@gmail.com"
  }
}
```

Errors

- 400 Bad Request - Invalid input
- 409 Conflict - Email already exists
- 500 Internal Server Error - Server side error



17.18.5 Common Authentication Methods

Method	Description
API Key	Client sends a key in header or query parameter. e.g., x-api-key: abc123
Bearer Token (JWT)	Token is sent in Authorization header. e.g., Authorization: Bearer <token>
Basic Auth	Username & password encoded in header. e.g., Authorization: Basic <base64>
OAuth 2.0	Use access token obtained via authorization flow.
API Key in Query	Key passed in URL query parameter. e.g., ?api_key=abc123



17.18.6 Documentation Best Practices

- * Keep documentation clear, simple and up to date.
- * Provide real examples for each endpoint.
- * Use consistent naming and structure.
- * Explain all parameters, responses and errors.
- * Include authentication details and rate limits.
- * Test all examples before publishing.
- * Use tools like Swagger (OpenAPI) for interactive docs.



17.18.7 Popular Tools for API Documentation

Swagger / OpenAPI

- Industry standard (OpenAPI)
- Auto-generate docs
- Interactive UI

Postman

- Create and share docs
- Mock servers
- Easy collaboration

ReadMe

- Beautiful docs site
- Versioning
- Markdown based

ReDoc

- Clean and simple UI
- Open source
- Easy to integrate

MkDocs + Markdown

- Static site generator
- Markdown based
- Highly customizable

Good documentation is as important as building a good API.

Key Takeaway:



Great API documentation makes it **easy** for developers to understand, integrate and use your API. It **reduces support**, **saves time** and helps your **API succeed**.

