Karyvio
LEARN | BUILD | GROW | GET HIRED

Express.js Complete Notes

Beginner to Advanced

A fast, minimalist and flexible web framework for Node.js

Small Steps Everyday Make You a Better Developer! ☺

1 What is Express.js ?

Ans →

- Express.js is a fast, minimalist and flexible web framework for Node.js.
- It simplifies the process of building web applications and APIs.
- It provides a set of powerful features for routing, middleware, handling requests and responses.
- It is minimal, which means it doesn't force a specific project structure, giving you full flexibility.
- Express.js is one of the most popular frameworks in the Node.js ecosystem.

Example : A Simple Express Server

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
    res.send('Hello from Express.js!');
});

app.listen(port, () => {
    console.log(`Server running at http://localhost:${port}`);
});
```

Output :

```
$ node server.js
Server running at http://localhost:3000
```

Open in browser http://localhost:3000
You will see: Hello from Express.js!

2 Why Express.js ?

Ans →

- Makes web development with Node.js easier.
- Simple and easy to learn.
- Lightweight and unopinionated.
- Provides powerful routing and middleware support.
- Large community and ecosystem.
- Flexible - you can use it for small projects or large-scale applications.
- Widely used in production by top companies.

Key Features :

- Minimal and lightweight
- Routing system
- Middleware support
- Flexible and unopinionated
- Easy integration with databases
- Large ecosystem (npm)
- Active community support
- Suitable for both small and large apps

Popular Extensions :

- express.Router()
- express.json()
- express.urlencoded()
- cookie-parser
- cors
- morgan (logging)
- helmet (security)
- and many more...

3 History of Express.js

Ans →

- Express.js was first released in 2010 by TJ Holowaychuk.
- It was created to simplify web application development with Node.js.
- It quickly gained popularity due to its simplicity and flexibility.
- Express.js is now maintained by the OpenJS Foundation and a large community of contributors.
- It has become the de facto standard web framework for Node.js.

Timeline :

2010	-	First release by TJ Holowaychuk
2011	-	Gained popularity in the Node.js community
2014	-	Express 4.0 released
2016	-	Moved to OpenJS Foundation
2020	-	Still one of the most used Node.js frameworks
2024	-	Continues to be actively maintained

"Simple ideas can make a big impact!"

4 Express.js vs Node.js

Ans →

Feature	Node.js	Express.js
What it is	Runtime environment	Web framework (built on Node.js)
Purpose	Runs JavaScript on server	Simplifies building web apps & APIs
Provides	Core modules (fs, http, etc.)	Routing, middleware, utilities
Level	Lower level	Higher level
Flexibility	You build everything yourself	Provides structure & helpful features
Learning curve	Moderate	Easy to learn
Examples	Node.js	Express.js (uses Node.js)

Relationship :

Express.js runs on top of Node.js. You need Node.js installed to use Express.js.

Express.js
↑ Built on top of
Node.js

5 Where is Express.js Used ?

Ans →

- Web applications (e.g., dashboards, CRM tools)
- RESTful APIs (for web and mobile apps)
- Backend for single-page applications (SPAs)
- Real-time applications (with WebSockets)
- Microservices and serverless functions
- Used by top companies like Netflix, Uber, LinkedIn, IBM, etc.

Real World Examples :

- Netflix - Uses Node.js and Express
- Uber - Uses Express for backend services
- LinkedIn - Uses Express in some of its services
- IBM - Uses Express in developer tools
- Many startups and SaaS products

Build Real Projects Gain Experience Grow Your Career with Karyvio ! ☺

Karyvio
LEARN | BUILD | GROW | GET HIRED

Express.js Setup & Installation

Good Setup Leads to Great Projects !

Get your Express.js environment ready and create your first project

1 Prerequisites

Ans →

- Node.js (v18 or higher)
- npm (comes with Node.js)
- A code editor (VS Code recommended)
- Basic knowledge of terminal / command line
- Basic understanding of JavaScript (ES6+)

Check Versions :

```
node -v     # e.g. v20.11.0
npm -v      # e.g. 10.2.4
npx -v      # e.g. 10.2.4
```

Make sure Node.js is installed before proceeding !

2 Create a New Project

Ans →

1. Create a project folder

```
mkdir express-app
cd express-app
```

2. Initialize a new Node.js project

```
npm init -y
```

3. Install Express.js

```
npm install express
```

4. (Optional) Install nodemon for development

```
npm install -D nodemon
```

Explanation :

- **npm init -y** creates a package.json file with default settings.
- **npm install express** installs the latest Express.js library in your project.
- **nodemon** automatically restarts the server when you make changes (used only in development).

Useful Tip :

Use nodemon to save time during development!

3 Project Structure

Ans →

A clean and organized folder structure :

```
express-app/
├── node_modules/     # installed packages
├── src/
│   ├── routes/       # route files
│   ├── controllers/  # request handlers
│   ├── middlewares/  # custom middleware
│   ├── app.js        # main application file
│   └── .env          # environment variables (optional)
├── package.json      # project metadata
└── package-lock.json # dependency lock file
```

Why This Structure ?

- Keeps your code modular and maintainable.
- Easy to scale for larger applications.
- Separates routes, controllers and middlewares.
- Commonly used in real-world Express projects.

package.json (important fields)

```
{
  "name": "express-app",
  "version": "1.0.0",
  "main": "src/app.js",
  "scripts": {
    "start": "node src/app.js",
    "dev": "nodemon src/app.js"
  },
  "dependencies": {
    "express": "^4.19.2"
  },
  "devDependencies": {
    "nodemon": "^3.1.0"
  }
}
```

Scripts Usage :

- **npm start** → Run the server
- **npm run dev** → Run with nodemon (auto-restart)

4 Install Additional Useful Packages

Ans →

Some commonly used packages in Express projects :

Package	Purpose
dotenv	Load environment variables from .env file
morgan	HTTP request logging
cors	Enable Cross-Origin Resource Sharing
helmet	Security middleware
express-validator	Input validation
cookie-parser	Parse cookies from requests

5 Next Step

Ans →

- Now that Express.js is installed and your project is set up, the next page will cover how to create your first Express server.
- We will write a simple server, run it, and understand how it works.

"Setup is the first step towards building something amazing !"

Karyvio
LEARN | BUILD | GROW | GET HIRED

Express.js

Creating Your First Server

Step by step setup and run your Express.js application

Practice
Build
Improve
Become a
Better Developer !

Let's install Express.js, create a simple server and understand how it works.

1 Create a Project Folder

Ans →
- Create a new folder for your project.
- Open it in your code editor (VS Code).
- Open the terminal inside this folder.

```
# Create and enter project folder                Terminal
mkdir express-app
cd express-app
```

Project Folder (Empty at this stage)

express-app/

(We will add files step by step)

2 Initialize a Node.js Project

Ans →
- This creates a package.json file to manage your project dependencies.
- You can press Enter for all default options.

```
# Initialize npm project                         Terminal
npm init -y
```

What is package.json ?
- It stores metadata about your project (name, version, dependencies, etc.).
- It helps in managing packages and scripts.
- It is automatically created when you run npm init.

Note :

-y flag automatically accepts all default options.
You can also run npm init without -y and fill the details manually.

3 Install Express.js

Ans →
- Now install Express.js using npm.
- This will add Express to your node_modules and also in package.json (dependencies).

```
# Install Express.js                             Terminal
npm install express
```

Check Installation
- After installation, you will see a node_modules folder in your project.
- You can also verify in package.json under "dependencies".

```
# Check installed packages                       Terminal
npm list express
```

4 Create Your First Express Server

Ans →
- Create a file named server.js in the root folder.
- Write a simple Express server code.

server.js

```
1  const express = require('express');
2  const app = express();
3  const port = 3000;
4
5  app.get('/', (req, res) => {
6    res.send('Hello from Express.js! 🚀');
7  });
8
9  app.listen(port, () => {
10   console.log(`Server running at http://localhost:${port}`);
11 });
```

Code Explanation :

1. Import the express module.
2. Create an Express application using express().
3. Define a port number (3000).
4. Create a route for the home page ("/").
5. Send a response to the browser.
6. Start the server using app.listen().
7. Log a message in the console.

5 Run the Server

Ans →
- Now start your server using Node.js.
- Open your browser and visit http://localhost:3000
- You should see the message: Hello from Express.js!

```
# Run the server                                 Terminal
node server.js
```

Output :

```
Server running at http://localhost:3000
```

Open in browser you will see this !

You just created your first Express.js server !
Keep going !

"A small setup today, a big opportunity tomorrow !"

Express.js

Express Application Object

One App Object Many Possibilities Build Amazing Apps !

Understand the Express application object (app) and its important methods.

1 What is Express Application Object ?

Ans →

- The application object (app) is an instance of an Express application.
- It is created by calling the express() function.
- The app object is the core of an Express.js application.
- It provides many methods and properties to configure middleware, routes, settings, and more.
- Using the app object, you define how your application will handle requests and responses.

Creating the App Object

Syntax :

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

Example :

```
const express = require('express');
const app = express();
console.log(typeof app); // object
```

Note :
express() is a function that returns an application object (app).

2 Important Methods of the App Object

Ans →

Method	Description
app.use()	Use middleware functions
app.get()	Handle GET requests
app.post()	Handle POST requests
app.put()	Handle PUT requests
app.patch()	Handle PATCH requests
app.delete()	Handle DELETE requests
app.listen()	Start the server
app.set()	Set application settings
app.get()	Get application settings
app.disable()	Disable a setting
app.enable()	Enable a setting

3 Using app.use()

Ans →

- app.use() is used to mount middleware functions.
- Middleware functions are executed for every incoming request.
- You can also use it to mount routers or serve static files.

Example : Using Middleware

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

app.use(express.json()); // built-in middleware
app.use((req, res, next) => {
    console.log('Request received...');
    next();
});
```

Tip :
Middleware runs in the order it is added using app.use().

4 Using app.listen()

Ans →

- app.listen() starts the Express server and makes it listen for incoming requests on a specified port.
- It takes a port number and a callback function (optional).

Example :

```
const app = express();
const port = 3000;

app.listen(port, () => {
    console.log('Server running at http://localhost:${port}');
});
```

Note :
The callback function is executed once the server is successfully started.

5 Application Settings with app.set()

Ans →

- You can use app.set() to set configuration values for your application.
- Common settings include the view engine, port, and environment.

Example :

```
const app = express();

app.set('port', 3000);          // set custom port
app.set('view engine', 'ejs'); // set template engine

console.log(app.get('port')); // 3000
```

You can retrieve values using app.get().

6 Putting It All Together

Ans → Here's a complete example using the Express application object :

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

app.use(express.json());
app.get('/', (req, res) => {
    res.send('Hello from Express app object!');
});

app.listen(3000, () => {
    console.log('Server running at http://localhost:3000');
});
```

Output :

```
Server running at
http://localhost:3000
```

Open this URL in your browser and you will see:

Hello from Express app object!

7 Key Takeaways

Ans →

- The app object is the core of an Express.js application.
- It is created using express().
- It provides methods to configure middleware, routes, and settings.
- app.use() is used for middleware.
- app.listen() starts the server.
- app.set() and app.get() are used for application settings.

"The app object gives you the power to build, configure and control your Express.js app!"

Karyvio
LEARN | BUILD | GROW | GET HIRED

Express.js

Request & Response Objects

Understand Requests Send Better Responses Build Better APIs !

Understand the req and res objects in Express.js and how to use them effectively

1 What are req and res Objects ?

Ans →

- In Express.js, every route handler function receives two objects: **req (request)** and **res (response).**
- These objects are provided by Express.js and contain useful information and methods.
- You can use them to read client data and send a response back to the client.

Basic Syntax :

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

app.get('/', (req, res) => {
   res.send('Hello!');
});
```

Note :
req and res are automatically available in every route handler function.

2 The Request Object (req)

Ans →

- The req object contains information about the incoming HTTP request from the client.
- It includes details like parameters, query, body, headers, cookies, method, URL, etc.

Common Properties of req Object :

Property	Description
req.params	Route parameters (e.g. /users/:id)
req.query	Query parameters (e.g. ?page=1)
req.body	Request body data (requires middleware)
req.headers	Request headers (e.g, content-type)
req.method	HTTP method (GET, POST, etc.)
req.url	Requested URL
req.ip	Client's IP address
req.cookies	Cookies (requires cookie-parser)
req.originalUrl	Original requested URL

Example : Using req Object

```
app.get('/users/:id', (req, res) => {
   console.log(req.params.id);   // route param
   console.log(req.query.page);  // query param
   console.log(req.headers['user-agent']); // header
   console.log(req.ip);      // client IP
   res.send('Request data received!');
});
```

3 The Response Object (res)

Ans →

- The res object is used to send a response back to the client.
- It provides many methods to send data in different formats like text, JSON, HTML, files, etc.

Common Methods of res Object :

Method	Description
res.send()	Send a response (string, object, HTML)
res.json()	Send a JSON response
res.status()	Set HTTP status code
res.sendFile()	Send a file
res.redirect()	Redirect to another URL
res.render()	Render a template (e.g, EJS, Pug)
res.cookie()	Set a cookie
res.end()	End the response

4 Example : Sending Different Responses

Ans →

```
app.get('/example', (req, res) => {
   res.send('This is a text response');
   // JSON response
   res.json({ message: 'Hello', success: true });
   // Set status code
   res.status(201).send('Created successfully!');
   // Redirect
   res.redirect('/about');
});
```

5 Chaining Methods

Ans →

- You can chain multiple methods together.
- Commonly used: res.status().json()

Example :

```
res.status(200).json({
   message: 'Data fetched successfully',
   data: { id: 1, name: 'Karyvio' })
});
```

6 Real World Use Case

Ans → Sending a success and error response :

```
app.get('/data',(req, res) => {
   const data ={ id: 1, name: 'Karyvio' };
   if (data) {
      return res.status(200).json({
         success: true,
         data
      });
   } else {
      return res.status(404).json({
         success: false,
         message: 'Data not found'
      });
   }
});
```

7 Key Takeaways

Ans →

- req contains information about the client request.
- res is used to send a response back to the client.
- req and res are provided by Express.js.
- Use req properties to read data and res methods to send data.
- These objects are essential for building APIs and web applications.

"Handle requests smartly, send responses effectively !"

Karyvio
LEARN | BUILD | GROW | GET HIRED

Express.js JS HTTP Methods

Different Methods
Different Purposes
Same Goal
Better APIs !

Learn about different HTTP methods in Express.js and when to use each of them.

1 What are HTTP Methods ?

Ans →

- HTTP methods (also called request methods) define the type of action to be performed on a resource.
- In Express.js, we use these methods to handle different types of requests.
- Each method has a specific purpose like fetching, creating, updating or deleting data.

2 Common HTTP Methods

Ans →

- The most commonly used HTTP methods are:

GET POST PUT PATCH DELETE

- Express.js provides a separate method for each of these (e.g. app.get(), app.post(), etc).
- You can also handle other methods like HEAD, OPTIONS if needed.

3 List of HTTP Methods and their Purpose

Ans →

Method	Purpose	Typical Use Case
GET	Fetch data	Get a list of users, get details
POST	Create data	Create a new user, submit form
PUT	Update data (full)	Update an entire resource
PATCH	Update data (partial)	Update only specific fields
DELETE	Delete data	Delete a resource
HEAD	Get response headers	Check if a resource exists
OPTIONS	Get allowed methods	Check which methods are allowed

Important Note :

- GET should be used only to fetch data. It should not modify data.
- POST is used to create new data.
- PUT replaces the entire resource.
- PATCH updates only specific fields.
- DELETE removes the resource.
- HEAD and OPTIONS are less common but useful in some cases (like CORS).

4 Code Examples for Each Method

Ans →

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

// GET - Fetch data
app.get('/users', (req, res) => {
    res.send('Get all users');
});

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

// PUT - Update entire data
app.put('/users/:id', (req, res) => {
    res.send('Update user with id ${req.params.id}');
});

// PATCH - Update partial data
app.patch('/users/:id', (req, res) => {
    res.send('Partial update for user ${req.params.id}');
});

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

5 Testing the Methods

Ans →

- You can test these methods using:
  - Postman
  - Thunder Client (VS Code extension)
  - cURL (command line)
  - Or a simple HTML form (for GET and POST)

Example using cURL :

```
# GET request
curl http://localhost:3000/users
# POST request
curl -X POST http://localhost:3000/users \
     -H "Content-Type: application/json" \
     -d '{"name": "Sovit"}'
```

6 Summary

Ans →

- HTTP methods define the type of operation.
- Express.js provides easy-to-use methods for each HTTP method.
- Use the correct method for the correct purpose to follow REST best practices.
- You can test the methods using Postman, cURL, or a browser.

7 Next Page

Ans →

- In the next page, we will learn about Routing in Express.js in detail.

"Use the right method, build the right API !"

Express.js

Routing in Express.js

Learn about routing in Express.js and how to organize your routes effectively.

Good Routes Make Your Application Clean, Scalable and Easy to Maintain !

1 What is Routing ?

Ans →

- Routing in Express.js is used to define how the application responds to different HTTP requests (based on URL and method).
- Each route is associated with a handler function that is executed when the route matches.
- It helps to organize your application into multiple endpoints (like /, /users, /products, etc.).
- Routing is a core feature of Express.js and is used to build REST APIs and web applications.

Simple Route Example :-

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

app.get('/', (req, res) => {
  res.send('Home Page');
});

app.get('/about', (req, res) => {
  res.send('About Page');
});

app.listen(3000, () => {
  console.log('Server running at http://localhost:3000');
});
```

Output :-

```
GET /      → Home Page
GET /about → About Page
```

2 Route Syntax

Ans →

```
app.METHOD(PATH, HANDLER)
```

- METHOD : HTTP method (get, post, put, etc.)
- PATH : URL path (e.g. /, /users, /about)
- HANDLER : Function to be executed when the route matches.

3 Example : Multiple Routes

Ans →

```
app.get('/', (req, res) => {
  res.send('Home Page');
});
app.get('/about', (req, res) => {
  res.send('About Page');
});
app.get('/contact', (req, res) => {
  res.send('Contact Page');
});
```

Output :-

```
GET /      → Home Page
GET /about → About Page
GET /contact → Contact Page
```

4 Route Handlers

Ans →

- A route handler is a function that gets executed when a route matches.
- You can define the handler function directly or use a separate function.

Example : Using a separate handler function

```
function homeHandler(req, res) {
  res.send('Welcome to Karyvio!');
}

app.get('/', homeHandler);
```

You can also use arrow functions :

```
app.get('/', (req, res) => {
  res.send('Hello!');
});
```

5 Using express.Router()

Ans →

- For larger applications, it is better to use express.Router() to organize routes into separate files (route modules).
- It helps to keep your code clean and maintainable.

Example :

```
const userRouter = require('./routes/user');
app.use('/users', userRouter);
```

All routes defined in userRouter will be prefixed with /users.

6 Routes with Different Methods

Ans →

- The same path can have different handlers for different HTTP methods.

Example :

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

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

Output :-

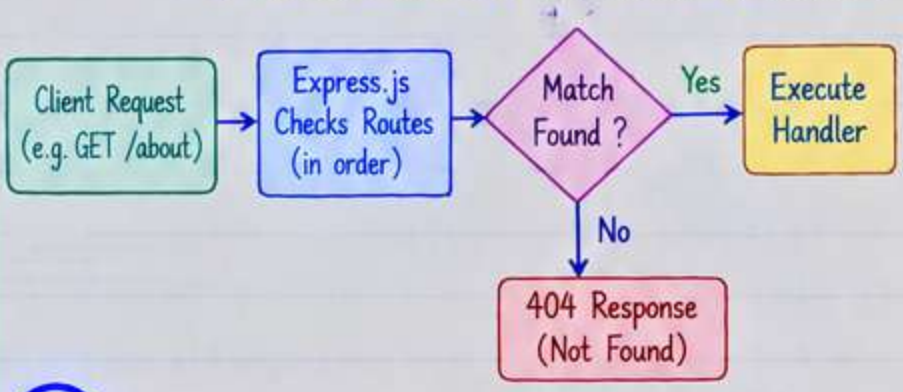
```
GET /users
  → Get all users

POST /users
  → Create a new user
```

7 Route Flow

Ans →

- When a request comes, Express.js checks the routes in the order they are defined.
- The first matching route is executed.
- If no route matches, a 404 handler is executed.



8 Key Takeaways

Ans →

- Routing is used to define how your application responds to different URLs and HTTP methods.
- Use app.get(), app.post(), app.put(), app.patch(), app.delete() to define routes.
- For better organization, use express.Router().
- Routes are checked in the order they are defined.
- If no route matches, a 404 response is returned.

"Well organized routes make your app scalable and easy to maintain !"

Karyvio
LEARN | BUILD | GROW | GET HIRED

Express.js

Route Parameters & Query Parameters

Small Concepts Big Possibilities Keep Learning !

Learn about route parameters and query parameters in Express.js and how to use them with examples.

1 What are Route Parameters ?

Ans →

- Route parameters are parts of the URL used to capture dynamic values.
- They are defined using a colon (:) before the parameter name.
- You can access the value using req.params.
- Useful when you want to identify a specific resource (e.g. user ID, product ID).

2 Syntax

Ans →

```
app.get('/route/:paramName', (req, res) => {
   // access using req.params.paramName
});
```

Note :
You can use multiple parameters in a single route.

3 Example : Single Route Parameter

Ans →

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

Request URL :
http://localhost:3000/users/101

Output :
User ID is 101

Key Point :
- :id is a route parameter.
- req.params.id contains the value from the URL.

4 Example : Multiple Route Parameters

Ans →

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

Request URL :
http://localhost:3000/users/5/posts/20

Output :
User ID: 5, Post ID: 20

Note :
You can use any name for the parameter (e.g. :id, :name, :postId, etc.)

5 What are Query Parameters ?

Ans →

- Query parameters are key-value pairs added after ? in the URL.
- They are used to pass optional data to the server.
- You can access them using req.query.
- Commonly used for filtering, searching, sorting, pagination, etc.

6 Syntax

Ans →

```
app.get('/route', (req, res) => {
   // access using req.query.paramName
});
```

7 Example : Using Query Parameters

Ans →

```
app.get('/products', (req, res) => {
   const search = req.query.search;
   const page = req.query.page;
   res.send(`Search: ${search}, Page: ${page}`);
});
```

Request URL :
http://localhost:3000/products?search=watch&page=2

Output :
Search: watch, Page: 2

Note :
Query parameters are always returned as strings. You can convert them to number if needed.

8 Real World Use Cases

Ans →

- Route Parameters :
  - Get user profile (/users/:id)
  - View product details (/products/:id)
  - Show blog post (/blog/:slug)
- Query Parameters :
  - Search products (/products?search=phone)
  - Pagination (/products?page=1&limit=10)
  - Filter data (/jobs?location=remote&type=internship)

9 Key Takeaways

Ans →

- Use route parameters for required, specific data (e.g. IDs, slugs).
- Use query parameters for optional data (e.g. search, filters, pagination).
- Access route parameters using req.params.
- Access query parameters using req.query.
- Both are very useful for building real world APIs.

"Dynamic routes and query parameters make your application flexible and powerful !"

Karyvio
LEARN | BUILD | GROW | GET HIRED

Express.js

Request Body & express.json()

Accept Data from Users and Build Dynamic Applications !

Learn how to receive data from the client using request body in Express.js and how to use express.json() middleware.

1 What is Request Body ?

Ans →

- The request body contains the data sent by the client to the server.
- It is commonly used in POST, PUT and PATCH requests.
- For example, when a user submits a form or sends JSON data from a frontend, the data comes in the request body.

2 What is express.json() ?

Ans →

- express.json() is a built-in middleware in Express.js.
- It parses incoming JSON data and makes it available in req.body.
- Without this middleware, req.body will be undefined.
- It should be added before your routes.

3 Basic Example : Using express.json()

Ans →

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

// Middleware to parse JSON data
app.use(express.json());

app.post('/user', (req, res) => {
  console.log(req.body);
  res.send('Data received successfully!');
});

app.listen(3000, () => {
  console.log('Server running at http://localhost:3000');
});
```

Send JSON (using Postman)

Method : POST
URL : http://localhost:3000/user
Headers :
Content-Type: application/json
Body (raw JSON) :

```
{
  "name": "Sovit",
  "email": "sovit@example.com",
  "age": 21
}
```

Output (in Terminal)

```
{
  name: 'Sovit',
  email: 'sovit@example.com',
  age: 21
}
Data received successfully!
```

Note :

- Always set the Content-Type header to application/json when sending JSON data.
- express.json() only parses JSON data, not form data or other types.

4 Example : Handling User Registration

Ans →

```
app.post('/register', (req, res) => {
  const { name, email, password } = req.body;
  if (!name || !email || !password) {
    return res.status(400).send('All fields are required!');
  }
  console.log(`User: ${name}, Email: ${email}`);
  res.send('User registered successfully!');
});
```

5 Accessing Nested Data

Ans → If the client sends nested JSON data, you can access it directly from req.body.

```
app.post('/profile', (req, res) => {
  const { user } = req.body;
  console.log(user.name);    // Access nested data
  console.log(user.address.city);
  res.send('Profile received!');
});
```

6 Other Body Parsers (for different data types)

Ans →

- express.json() ⇒ Parses JSON data (used in modern apps)
- express.urlencoded() ⇒ Parses URL-encoded data (form data)
- express.text() ⇒ Parses plain text data
- express.raw() ⇒ Parses raw binary data

Example : Using express.urlencoded() (for form data)

```
app.use(express.urlencoded({ extended: true }));

app.post('/submit', (req, res) => {
  console.log(req.body); // { name: 'Sovit', email: 'test@example.com' }
  res.send('Form data received!');
});
```

7 Common Mistakes

Ans →

- Forgetting to use express.json()
- Not setting Content-Type: application/json
- Using req.body before adding the middleware
- Trying to parse non-JSON data with express.json()
- Placing middleware after routes

8 Key Takeaways

Ans →

- Request body is used to receive data from client.
- Use express.json() to parse JSON data.
- Always set the correct Content-Type header.
- You can also use express.urlencoded() for form data.
- Access data easily using req.body.
- Essential for building APIs and handling user input.

" Handle incoming data safely, validate it properly, and build secure applications ! "

Karyvio
LEARN | BUILD | GROW | GET HIRED

Express.js

Static Files & express.static()

Static files help you serve frontend assets like HTML, CSS, JS and images easily in Express.js !

Learn how to serve static files (HTML, CSS, JavaScript, images, etc.) using express.static() middleware in Express.js.

1 Ans → What are Static Files ?

- Static files are files that do not change frequently.
- Examples: HTML, CSS, JavaScript, images, fonts, PDFs, etc.
- These files are sent to the browser as they are, without any processing by the server.
- They are commonly used to build the frontend part of a web application.

2 Ans → What is express.static() ?

- express.static() is a built-in middleware in Express.js.
- It is used to serve static files from a directory.
- When a client requests a file, Express looks for it in the specified directory and sends it to the browser.

3 Ans → Basic Syntax

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

// Serve static files from 'public' directory
app.use(express.static('public'));

app.listen(3000, () => {
  console.log('Server running at http://localhost:3000');
});
```

Now, all files inside the 'public' folder will be accessible in your browser.

4 Ans → Project Folder Structure

```
my-app/
├── public/
│   ├── index.html
│   ├── style.css
│   ├── script.js
│   └── images/
│       └── logo.png
├── app.js
└── package.json
```

All files inside the public folder will be served as static files.

5 Ans → Example : Serving HTML, CSS, JavaScript and Images

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

app.use(express.static('public'));

app.listen(3000, () => {
  console.log('Visit http://localhost:3000');
});
```

Access in Browser:

- http://localhost:3000/index.html
- http://localhost:3000/style.css
- http://localhost:3000/script.js
- http://localhost:3000/images/logo.png

6 Ans → Serving Files from a Custom Path

```
app.use(express.static('public'));

// Or use an absolute path
const path = require('path');
app.use(express.static(path.join(__dirname,
    'public')));
```

Using path.join(__dirname, 'public') is safer, especially in larger projects.

7 Ans → Mounting Static Files under a URL Prefix

```
app.use('/assets', express.static('public'));

// Now files are accessed like:
// http://localhost:3000/assets/style.css
// http://localhost:3000/assets/images/logo.png
```

Useful when you want to keep your static files under a specific route.

8 Ans → Multiple Static Directories

```
app.use(express.static('public'));
app.use(express.static('uploads'));
app.use('/assets', express.static('assets'));

// Express will look for files in all specified
directories in the order they are added.
```

9 Ans → Common Mistakes

- Wrong folder path (e.g. 'Public' instead of 'public').
- Placing express.static() after your routes (static files might not work).
- Expecting dynamic processing (use templates like EJS, not static).
- Forgetting to set the correct file path when using path.join().
- Exposing sensitive files (e.g. .env) in the public folder.

10 Ans → Key Takeaways

- express.static() helps you serve static files easily.
- Keep your static files (HTML, CSS, JS, images) in a separate folder like public.
- You can mount static files under a custom path.
- You can use multiple static directories.
- Always use path.join() for better path handling.
- Do not put sensitive files in the public folder.

"Static files make your web app look great !"

Express.js

Middleware in Express.js (Fundamentals)

Middleware sits between the request and response, allows you to run code, modify the request/response or end the request cycle. It's the heart of Express.js !

Learn what middleware is in Express.js, how it works, and how to use it to process requests and responses.

1 Ans → What is Middleware ?

- Middleware is a function that gets executed during the request-response cycle.
- It has access to the request (req), response (res) objects and the next() function.
- It can execute any code, modify req or res, end the request-response cycle, or call the next middleware in the stack.

2 Ans → How Middleware Works ?

- When a request comes to your Express application, it passes through the middleware functions in the order they are defined.
- Each middleware can perform some logic and then either end the request or call next() to pass control to the next middleware.

3 Ans → Basic Syntax

```
const middlewareFunction = (req, res, next) => {
  // do something
  next(); // pass control to the next middleware
};

app.use(middlewareFunction);
```

Middleware always has three parameters: req, res, next

4 Ans → Example : Simple Logging Middleware

```
const logger = (req, res, next) => {
  console.log(`${new Date().toISOString()} - ${req.method} ${req.url}`);
  next();
};

app.use(logger);
```

5 Ans → Multiple Middleware Functions

You can use multiple middleware functions:

```
const m1 = (req, res, next) => {
  console.log('Middleware 1');
  next();
};
const m2 = (req, res, next) => {
  console.log('Middleware 2');
  next();
};
app.use(m1, m2);
```

Middleware execute in the order they are added.

6 Ans → Middleware Use Cases

- Logging requests
- Authentication and authorization
- Parsing request bodies
- Handling errors
- Serving static files
- Modifying request or response objects
- Rate limiting
- Input validation
- Any custom logic before reaching your route handlers

7 Ans → Global vs Route-level Middleware

Global Middleware (applies to all routes)

```
app.use((req, res, next) => {
  console.log('Global middleware');
  next();
});
```

Route-level Middleware (applies to specific routes)

```
app.get('/user', (req, res, next) => {
  console.log('Route middleware');
  next();
}, (req, res) => {
  res.send('User page');
});
```

8 Ans → Ending the Request in Middleware

If you don't call next(), the request will not move to the next middleware or route.

```
app.use((req, res, next) => {
  if (req.query.block) {
    res.status(403).send('Blocked!');
    return; // next() not called
  }
  next();
});
```

9 Ans → Common Mistakes

- Forgetting to call next() (request gets stuck).
- Not using middleware in the correct order.
- Modifying res after it has already been sent.
- Doing heavy operations in middleware (use async or background jobs).
- Using middleware when a simple route handler is enough.

10 Ans → Key Takeaways

- Middleware is the core concept in Express.js.
- It has access to req, res and next().
- You can use it globally or for specific routes.
- It helps in building clean, modular and scalable applications.
- Middleware can modify request/response, end the request, or pass control to the next middleware.

"Middleware gives you the power to control the flow of your application !"

Express.js

Built-in Middleware in Express.js

Learn about the built-in middleware provided by Express.js and how to use them in your applications.

Express.js comes with several built-in middleware functions that help you handle common tasks easily without installing extra packages !

1 Ans → What are Built-in Middleware ?

- Built-in middleware are the middleware functions that come with Express.js by default.
- They are part of the Express core and do not require any separate installation.
- They help in handling common tasks like parsing request bodies, serving static files, handling URLs, etc.

2 Ans → List of Built-in Middleware

- express.json() ⇒ Parse JSON request bodies
- express.urlencoded() ⇒ Parse URL-encoded data
- express.static() ⇒ Serve static files
- express.text() ⇒ Parse text request bodies
- express.raw() ⇒ Parse raw (Buffer) data

3 Ans → express.json()

Parses incoming JSON request bodies and makes the data available in req.body.

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

app.use(express.json());

app.post('/api', (req, res) => {
   console.log(req.body); // JSON data
   res.send('Data received');
});
```

Use this when the client sends data with Content-Type: application/json

4 Ans → express.urlencoded()

Parses incoming URL-encoded data (from HTML forms) and makes it available in req.body.

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

app.use(express.urlencoded({ extended: true }));

app.post('/form', (req, res) => {
   console.log(req.body); // form data
   res.send('Form data received');
});
```

Use this for HTML form submissions with Content-Type: application/x-www-form-urlencoded

5 Ans → express.static()

Serves static files like HTML, CSS, JavaScript, images, etc.

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

app.use(express.static('public'));

// Now files in 'public' folder are accessible
// e.g. http://localhost:3000/index.html
```

Useful for serving frontend assets without writing custom code. (We already tested this in Page 10)

6 Ans → express.text()

Parses incoming request bodies as plain text.

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

app.use(express.text());

app.post('/text', (req, res) => {
   console.log(req.body); // plain text
   res.send('Text received');
});
```

Use this when the client sends plain text data with Content-Type: text/plain

7 Ans → express.raw()

Parses incoming request bodies as a Buffer (raw data).

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

app.use(express.raw());

app.post('/webhook', (req, res) => {
   console.log(req.body); // Buffer data
   res.send('Raw data received');
});
```

Useful for handling webhooks (like Stripe) where the raw request body is needed for signature verification.

8 Ans → When to Use Which ?

Middleware	Use Case
express.json()	Parse JSON data (APIs, modern apps)
express.urlencoded()	Parse form data (HTML forms)
express.static()	Serve static files (HTML, CSS, JS, images)
express.text()	Parse plain text data
express.raw()	Parse raw data (webhooks, binary data)

9 Ans → Important Notes

- These middleware functions must be used before your routes.
- Use only the middleware you need.
- express.json() and express.urlencoded() cannot parse files (for that, use multer).
- Middleware order matters in Express.js.

10 Ans → Key Takeaways

- Express.js provides useful built-in middleware.
- Use express.json() for JSON data.
- Use express.urlencoded() for form data.
- Use express.static() for serving static files.
- Use express.text() and express.raw() for special cases.
- They are easy to use and built into Express.js.

"Use the right middleware, make your app work smarter!"

Express.js

Custom Middleware in Express.js

Custom middleware allows you to run your own code during the request-response cycle in Express.js !

Learn how to create and use your own middleware functions in Express.js to handle custom logic, validation, authentication, logging, and more.

1 Ans → What is Custom Middleware ?

- Custom middleware is a function that you define yourself to perform specific tasks during the request-response cycle.
- It gives you full control over what happens before the request reaches the route handler.
- You can use it for logging, authentication, validation, error handling, modifying request/response, etc.

2 Ans → Middleware Function Structure

```
function myMiddleware(req, res, next) {
  // do something
  next(); // pass control to the next middleware
}
```

A middleware function always has three parameters: req, res, next

3 Ans → Simple Example : Logging Middleware

```
const logger = (req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next(); // move to the next middleware
};

app.use(logger);

// or use for a specific route
app.get('/', logger, (req, res) => {
  res.send('Home Page');
});
```

This middleware logs the HTTP method and URL for every request.

4 Ans → Example : Authentication Middleware

```
const isAuthenticated = (req, res, next) => {
  const token = req.haaders.authorization;
  if (!token) {
    return res.status(401).json({ message: 'Unauthorized' });
  }
  // You can verify token here (e.g., JWT verification)
  console.log('User authenticated');
  next();
};

app.get('/dashboard', isAuthenticated, (req, res) => {
  res.send('Welcome to Dashboard');
});
```

This middleware checks if the user is authenticated before accessing the route.

5 Ans → Example : Request Validation Middleware

```
const validateUser = (req, res, next) => {
  const { name, email } = req.body;

  if (!name || !email) {
    return res.status(400).json({
      message: 'Name and email are required'
    });
  }
  next();
};

app.post('/users', validateUser, (req, res) => {
  res.json({ message: 'User created successfully' });
});
```

This middleware validates the request body before creating a user.

6 Ans → Example : Error Handling in Middleware

```
const errorHandler = (err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ message: 'Something went wrong!' });
};

app.use(errorHandler);
```

Error-handling middleware has 4 parameters: (err, req, res, next)

7 Ans → Use Cases for Custom Middleware

- Logging requests
- Authentication and authorization
- Validating request data (body, params, query)
- Handling errors
- Modifying request or response objects
- Rate limiting
- Tracking user activity
- Any custom logic specific to your application

8 Ans → Middleware Execution Order

Middleware functions are executed in the order they are added. If next() is not called, the request will not move to the next middleware or route handler.

9 Ans → Common Mistakes

- Forgetting to call next().
- Not handling errors properly.
- Not using middleware only where needed (performance issue).
- Modifying req or res incorrectly.
- Placing middleware in the wrong order.
- Using async middleware without proper error handling (use try-catch or async error handler).

10 Ans → Key Takeaways

- Custom middleware gives you flexibility to add your own logic.
- Always use next() to pass control to the next middleware.
- Use it for logging, validation, authentication, and more.
- Middleware order matters.
- Error-handling middleware has 4 parameters.
- Keep your middleware functions modular and reusable.

"Small middleware, big possibilities !"

Express.js

Router-Level Middleware in Express.js

Router-level middleware is applied to an Express Router instance and affects only the routes defined in that router, not the entire application.

Learn how to use router-level middleware in Express.js to apply middleware to a group of routes and keep your code modular and organized.

1 Ans → What is Router-Level Middleware ?

- Router-level middleware is middleware that is applied to a specific router instance using express.Router().
- It runs for all routes defined in that router.
- It does not affect routes defined in other routers or the main application (app).
- Useful for organizing middleware by feature (e.g., auth middleware for protected routes).

2 Ans → Why Use Router-Level Middleware ?

- Keep your code modular and organized.
- Apply middleware only to specific routes.
- Avoid cluttering the main app with too many middleware functions.
- Easier to maintain and scale your application.
- Commonly used for authentication, logging, validation, and authorization.

3 Ans → Basic Syntax

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

// Apply middleware to all routes in this router
router.use((req, res, next) => {
  console.log('Router-level middleware');
  next();
});

// Define routes
router.get('/', (req, res) => {
  res.send('Router Home');
});

module.exports = router;
```

Use express.Router() to create a new router and apply middleware using router.use().

4 Ans → Example : Protected Routes with Auth Middleware

```
const express = require('express');
const router = express.Router();
// Auth middleware
const authMiddleware = (req, res, next) => {
  const token = req.headers.authorization;
  if (!token) {
    return res.status(401).json({ message: 'Access denied' });
  }
  // You can verify the token here
  next();
};

// Apply auth middleware to all routes in
in this router
router.use(authMiddleware);

router.get('/profile', (req, res) => {
  res.json({ message: 'User profile', user: req.user });
});
module.exports = router;
```

All routes in this router will require a valid token.

5 Ans → Multiple Middleware in a Router

```
const router = express.Router();

const logger = (req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
};

const auth = (req, res, next) => {
  // auth logic here
  next();
};

// Apply multiple middleware
router.use(logger, auth);

router.get('/data', (req, res) => {
})res.json({ message: 'Protected data' });
```

You can apply multiple middleware functions in the router using router.use(mw1, mw2).

6 Ans → Applying Middleware to Specific Routes

```
const router = express.Router();
const isAdmin = (req, res, next) => {
  if (req.user?.role !== 'admin') {
    return res.status(403).json({ message: 'Admin access only' });
  }
  next();
};
// Apply middleware only to this route
router.get('/admin', isAdmin, (req, res) => {
  res.json({ message: 'Welcome Admin' });
});
// Other routes without this middleware
router.get('/public', (req, res) => {
  res.json({ message: 'This is a public route' });
});
```

Middleware can be applied to individual routes instead of the entire router.

7 Ans → Project Structure with Router-Level Middleware

```
my-app/
├── app.js
├── routes/
│   ├── userRoutes.js   (router-level middleware)
│   ├── adminRoutes.js  (router-level middleware)
│   └── publicRoutes.js (no middleware)
└── middleware/
    ├── auth.js
    └── logger.js
```

Each router can have its own middleware based on the requirement.

8 Ans → Router-Level vs App-Level Middleware

Router-Level Middleware	App-Level Middleware
Applied to a specific router	Applied to the entire application
Affects only routes in that router	Affects all routes
Used for modular code	Used for global tasks (e.g., logging)
Defined using router.use()	Defined using app.use()
Example: Auth for /admin	Example: JSON parser for all routes

9 Ans → Common Mistakes

- Forgetting to call next().
- Applying middleware in the wrong order.
- Expecting router-level middleware to work on other routers.
- Not handling errors inside middleware.
- Using too many middleware functions unnecessarily.

10 Ans → Key Takeaways

- Router-level middleware helps you keep your code modular.
- Use express.Router() to create a router.
- Apply middleware using router.use().
- It only affects the routes defined in that router.
- Useful for authentication, validation, and route-specific logic.
- Keep your middleware focused and reusable.

Express.js

Error-Handling Middleware in Express.js

Error-handling middleware helps you catch and handle errors in a centralized way, preventing your app from crashing and providing meaningful responses.

Learn how to handle errors gracefully in Express.js using error-handling middleware, so your application is more reliable and user-friendly.

1 Ans → What is Error-Handling Middleware ?

- Error-handling middleware is a special type of middleware that catches errors in the request-response cycle.
- It has four parameters: (err, req, res, next).
- It is used to handle both synchronous and asynchronous errors in a centralized way.
- It helps you send a consistent and user-friendly error response instead of crashing the server.

2 Ans → Syntax of Error-Handling Middleware

```
const errorHandler = (err, req, res, next) => {
  // handle the error here
};

app.use(errorHandler);
```

Note: Error-handling middleware must have 4 parameters (err, req, res, next).

3 Ans → Example : Basic Error Handler

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({
    message: 'Something went wrong!',
    error: process.env.NODE_ENV === 'development'
      ? err.message
      : 'Internal Server Error'
  });
});
```

In development mode, we send the actual error message. In production, we send a generic message.

4 Ans → Handling 404 - Not Found

```
app.use((req, res, next) => {
  const error = new Error(`Cannot ${req.method} ${req.url}`);
  error.status = 404;
  next(error);
});

// Error handler (should be after all routes)
app.use((err, req, res, next) => {
  res.status(err.status || 500).json({
    message: err.message || 'Internal Server Error'
});
```

This will catch all undefined routes and send a 404 response.

5 Ans → Handling Async Errors

If you are using async/await in routes, you need to pass errors to next(err) because try-catch is not automatically handled.

```
app.get('/async', async (req, res, next) => {
  try {
    // some async code
    const data = await getData();
    res.json(data);
  } catch (err) {
    next(err); // pass error to error-handling middleware
  }
});
```

You can also use libraries like express-async-handler to handle async errors automatically.

6 Ans → Using Custom Error Classes

You can create custom error classes to handle different types of errors.

```
class AppError extends Error {
  constructor(message, statusCode) {
    super(message);
    this.statusCode = statusCode;
    this.status = `${statusCode}`.startsWith('4')
      ? 'fail'
      : 'error';
    Error.captureStackTrace(this, this.constructor);
  }
}
// Usage in a route
app.get('/profile', (req, res, next) => {
    return next(new AppError('User not found', 404));
});
```

Custom error classes make your error handling more structured and maintainable.

7 Ans → Error Response Format (Best Practice)

```
app.use((err, req, res, next) => {
  const statusCode = err.statusCode || 500;
  res.status(statusCode).json({
    success: false,
    message: err.message || 'Internal Server Error',
    ...(process.env.NODE_ENV === 'development' && {
      stack: err.stack
    })
  });
});
```

A consistent error response format helps in debugging and improves API usability.

8 Ans → Built-in Error Types

Some common built-in errors you may encounter:

Error Type	Description
SyntaxError	Invalid JSON in request body (handled by express.json())
ValidationError	Validation failed (from libraries like express-validator)
CastError	Invalid MongoDB ObjectId (when using Mongoose)
TypeError	Passing wrong type of argument
ReferenceError	Using an undefined variable

9 Ans → Common Mistakes

- Not using 4 parameters (err, req, res, next).
- Placing error-handling middleware before routes.
- Not handling async errors (not using next(err)).
- Sending sensitive error details in production.
- Forgetting to set proper status codes.

10 Ans → Key Takeaways

- Error-handling middleware catches errors in a centralized way.
- Always use 4 parameters (err, req, res, next).
- Handle both synchronous and asynchronous errors.
- Use custom error classes for better structure.
- Send consistent and secure error responses.
- Place error-handling middleware after all routes.

"Good error handling doesn't hide problems, it helps you solve them faster !"

Express.js

Express Router in Express.js

Express Router allows you to create modular route handlers in separate files and mount them in your main application. It helps keep your code clean and organized !

Learn how to use Express Router to create modular routes and keep your application organized, scalable, and maintainable.

1 Ans → What is Express Router ?

- Express Router is a mini version of the Express app.
- It allows you to define a group of routes in a separate module using express.Router().
- It helps in organizing routes by feature (e.g., users, products).
- It can have its own middleware and route handlers.
- It makes your application more modular and maintainable.

2 Ans → Basic Syntax

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

router.get('/', (req, res) => {
  res.send('Router is working!');
});
module.exports = router;
```

Use express.Router() to create a new router instance.

3 Ans → Example : Creating a Separate Route File

routes/userRoutes.js

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

router.get('/', (req, res) => {
  res.json({ message: 'Get all users' });
});

router.post('/', (req, res) => {
  res.json({ message: 'Create a new user' });
});

module.exports = router;
```

Define all user related routes in a separate file and export the router.

4 Ans → Using Router in the Main App

app.js (or index.js)

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

const userRoutes = require('./routes/userRoutes');

app.use(express.json());
app.use('/api/users', userRoutes); // mount router

app.get('/', (req, res) => {
  res.send('Express App is running!');
});
app.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

Mount the router using app.use() with a URL prefix.

5 Ans → Multiple Routers Example

routes/productRoutes.js

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

router.get('/', (req, res) => {
  res.json({ message: 'Get all products' });
});
module.exports = router;
```

You can create multiple route files for different features like users, products, orders, etc.

app.js

```
const productRoutes = require('./routes/productRoutes');
app.use('/api/products', productRoutes);
```

6 Ans → Router with Middleware

```
const express = require('express');
const router = express.Router();
// Route-specific middleware
router.use((req, res, next) => {
  console.log('Time:', new Date().toISOstring());
  next();
});

router.get('/', (req, res) => {
  res.json({ message: 'This route has its own middleware' });
});
module.exports = router;
```

You can use middleware specific to this router only.

7 Ans → Route Parameters and Router

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

router.get('/:id', (req, res) => {
  const userId = req.params.id;
  res.json({ message: 'User ID: ${userId}' });
});
module.exports = router;
```

Route parameters work the same way inside a router.

8 Ans → Folder Structure (Recommended)

```
my-app/
├── app.js
├── routes/
│   ├── userRoutes.js
│   ├── productRoutes.js
│   └── orderRoutes.js
├── controllers/
├── middleware/
└── models/
```

Keep all route files inside a 'routes' folder for better organization.

9 Ans → Common Mistakes

- Forgetting to export the router (module.exports = router).
- Not mounting the router in the main app using app.use().
- Incorrect route paths or prefixes.
- Defining too many routes in one file (keep it modular).
- Using router instead of app.use() in the main file.

10 Ans → Key Takeaways

- Express Router helps you create modular and scalable routes.
- Use express.Router() to define routes in separate files.
- Mount routers using app.use('/prefix', router).
- Routers can have their own middleware.
- It keeps your code clean, organized, and easy to maintain.
- Widely used in real-world Express.js applications.

"Modular routes, bigger possibilities !"

Express.js

Controllers & Project Structure in Express.js

Learn how to organize your Express.js application using controllers and a clean project structure for better maintainability and scalability.

A clean project structure with controllers helps you keep your code organized, modular, and easy to maintain as your Express.js application grows !

1 Ans → What are Controllers ?

- Controllers are functions that handle the business logic for your routes.
- They receive the request, process the data (e.g., interact with models or services), and send the response.
- Using controllers separates your route definitions from your application logic.
- It makes your code more modular, readable, and maintainable.

2 Ans → Why Use Controllers ?

- Keeps your routes clean and simple.
- Separates business logic from route definitions.
- Makes your application easier to maintain and scale.
- Helps with code reusability.
- Follows best practices and is widely used in real-world projects.

3 Ans → Example : Simple Controller

controllers/userController.js

```
// Controller function to get all users
const getUsers = async (req, res) => {
  try {
    // In a real app, fetch users from a database
    const users = [
      { id: 1, name: 'Sovit', email: 'sovit@example.com' },
      { id: 2, name: 'Smruti', email: 'smruti@example.com' }
    ];
    res.status(200).json({ message: 'Users fetched successfully',
      users });
  } catch (error) {
    console.error(error);
    res.status(500).json({ message: 'Server error' });
  }
};
module.exports = { getUsers };
```

Controllers handle the request logic and send the response.

4 Ans → Using Controller in a Route

routes/userRoutes.js

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

router.get('/', getUsers); // GET /api/users

module.exports = router;
```

Import the controller and use it in your routes.

5 Ans → Registering the Route in the Main App

app.js

```
const express = require('express');
const userRoutes = require('./routes/userRoutes');
const app = express();

app.use(express.json());
app.use('/api/users', userRoutes);

app.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

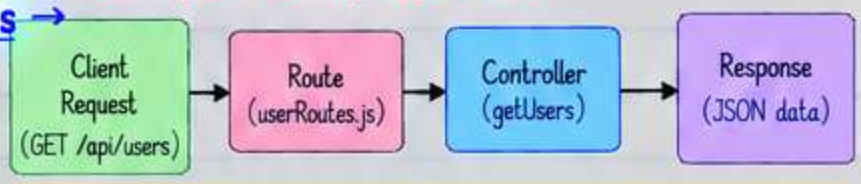
Mount the route with a prefix (e.g., /api/users).

6 Ans → Recommended Project Structure

```
my-app/
├── app.js                # Entry point
├── config/               # Configuration files (DB, env, etc.)
├── controllers/          # Controller functions
│   └── userController.js
├── routes/               # Route definitions
│   └── userRoutes.js
├── models/               # Database models (optional)
├── middleware/           # Custom middleware (optional)
├── utils/                # Helper functions (optional)
└── package.json
```

This structure keeps your code organized and scalable.

7 Ans → Example : Complete Flow



Request → Route → Controller → Response

Each part has a specific responsibility, making your code clean and modular.

8 Ans → Best Practices

- Keep controllers focused on business logic.
- Use async/await and proper error handling.
- Follow a consistent folder structure.
- Use meaningful file and function names.
- Separate concerns (routes, controllers, models, middleware).
- Use environment variables for configuration.
- Keep your code modular and reusable.

9 Ans → Common Mistakes

- Putting all logic inside route files.
- Not handling errors in controllers.
- Using controllers for simple tasks that belong in middleware.
- Not following a consistent folder structure.
- Mixing business logic with database queries (consider using services).
- Not using async/await for asynchronous operations.

10 Ans → Key Takeaways

- Controllers help you keep your Express.js application organized.
- Use a clean and scalable project structure.
- Separate routes, controllers, and models.
- Follow best practices for maintainability.
- A well-structured project makes it easier to add new features in the future.

Express.js

REST API Development in Express.js

A REST API allows different applications to communicate over HTTP using standard methods. Express.js makes it easy to build powerful and scalable REST APIs !

Learn how to build a complete REST API using Express.js, following best practices, proper structure, and real-world examples.

1 Ans → What is a REST API ?

- REST (Representational State Transfer) is an architectural style for building web APIs.
- It uses standard HTTP methods (GET, POST, PUT, PATCH, DELETE).
- Resources are identified using URLs.
- Data is exchanged in JSON format (usually).
- Stateless: each request from a client contains all the information needed.
- Widely used for web and mobile applications.

2 Ans → REST API Principles

- Use standard HTTP methods.
- Use meaningful and consistent URLs.
- Use proper HTTP status codes.
- Exchange data in JSON format.
- Keep the API stateless.
- Follow a consistent naming convention.
- Version your API (e.g., /api/v1).
- Provide clear and useful error messages.

3 Ans → Project Setup

```
mkdir express-api
cd express-api
npm init -y
npm install express
npm install nodemon --save-dev
```

Initialize a new Node.js project and install Express. We will use nodemon for development to automatically restart the server.

4 Ans → Basic Project Structure

```
express-api/
├── src/
│   ├── controllers/
│   ├── routes/
│   ├── models/
│   ├── middleware/
│   ├── utils/
│   ├── app.js
│   └── server.js
└── package.json
```

A clean folder structure helps keep your code modular, maintainable, and scalable.

5 Ans → Creating a REST API for Users

routes/userRoutes.js

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

const { getUsers, getUser, createUser, updateUser,
  deleteUser } = require('../controllers/userController');

router.get('/', getUsers);          // GET /api/users
router.get('/:id', getUser);        // GET /api/users/:id
router.post('/', createUser);       // POST /api/users
router.put('/:id', updateUser);     // PUT /api/users/:id
router.patch('/:id', updateUser);   // PATCH /api/users/:id
router.delete('/:id', deleteUser);  // DELETE /api/users/:id

module.exports = router;
```

Define routes for all CRUD operations on the users resource.

6 Ans → Controller Logic

controllers/userController.js

```
let users = [];
let id = 1;

exports.getUsers = (req, res) => {
  res.status(200).json(urers);
};

exports.getUser = (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));
  if (!user) return res.status(404).json({ message: 'User not found' });
  res.json(user);
};

exports.createUser = (req, res) => {
  const { name, email } = req.body;
  const newUser = { id: id++, name, email };
  users.push(newUser);
  res.status(201).json(newUser);
};
// similarly implement updateUser and deleteUser
```

Controllers contain the business logic for each route. In a real project, you would use a database instead of an in-memory array.

7 Ans → Connecting Routes in the App

app.js

```
const express = require('express');
const app = express();
const userRoutes = require('./routes/userRoutes');

app.use(express.json());
app.use('/api/users', userRoutes);
module.exports = app;
```

Use express.json() middleware to parse JSON request bodies and mount your routes with a prefix (e.g., /api/users).

8 Ans → Starting the Server

server.js

```
const app = require('./app');
const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
  console.log('Server running on http://localhost:${PORT}');
});
```

Start the Express server and make your API live.

9 Ans → Testing the API

- Use tools like Postman, Thunder Client, or curl to test your API.
- Example requests:

GET	http://localhost:3000/api/users
POST	http://localhost:3000/api/users (with JSON body)
GET	http://localhost:3000/api/users/1
PUT	http://localhost:3000/api/users/1
DELETE	http://localhost:3000/api/users/1

Test all CRUD operations to ensure your API works correctly.

10 Ans → Key Takeaways

- REST APIs use standard HTTP methods and JSON.
- Follow a modular project structure (routes, controllers, etc.).
- Use meaningful URLs and proper status codes.
- Keep your code clean and maintainable.
- Test your API thoroughly.
- Use a database (MongoDB, PostgreSQL, etc.) in real projects.
- Consider adding validation, error handling, and authentication in the next steps.

Karyvio
LEARN | BUILD | GROW | GET HIRED

Express.js

HTTP Status Codes & API Responses in Express.js

Learn about HTTP status codes, how to send proper API responses in Express.js, and follow best practices to make your APIs consistent, professional, and easy to use.

Good APIs communicate clearly. Use the right status codes and consistent responses to make your application professional, reliable, and easy to maintain !

1 Ans → What are HTTP Status Codes ?

- HTTP status codes are 3-digit numbers returned by the server to indicate the result of a request.
- They help the client understand whether the request was successful, failed, or needs further action.
- Using the correct status code improves API clarity and makes debugging easier.

2 Ans → Categories of HTTP Status Codes

1xx	Informational	(e.g., 100 Continue)
2xx	Success	(e.g., 200 OK)
3xx	Redirection	(e.g., 301 Moved Permanently)
4xx	Client Error	(e.g., 400 Bad Request)
5xx	Server Error	(e.g., 500 Internal Server Error)

3 Ans → Commonly Used Status Codes

Status Code	Meaning	When to Use
200	OK	Successful GET request
201	Created	Resource created (POST)
204	No Content	Successful request, no data to return
400	Bad Request	Invalid input / validation error
401	Unauthorized	Authentication required or failed
403	Forbidden	Access denied (no permission)
404	Not Found	Resource not found
409	Conflict	Resource already exists (e.g., duplicate)
422	Unprocessable Entity	Validation errors (detailed)
500	Internal Server Error	Something went wrong on the server

4 Ans → Sending Basic Responses

app.js

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

app.get('/', (req, res) => {
  res.status(200).json({
    message: 'Welcome to Karyvio!'
  });
});

app.get('/not-found', (req, res) => {
  res.status(404).json({
    error: 'Resource not found'
  });
});
```

Use res.status() to set the status code and res.json() to send the response.

5 Ans → Success Response Example (201 Created)

app.js

```
app.post('/users', (req, res) => {
  const { name, email } = req.body;
  // Save user to database (dummy example)
  const newUser = { id: 1, name, email };
  res.status(201).json({
    message: 'User created successfully';
    data: newUser
  });
});
```

Use 201 when a new resource is created (e.g., a new user).

6 Ans → Error Response Example (400 Bad Request)

app.js

```
app.post('/users', (req, res) => {
  const { name, email } = req.body;

  if (!name || !email) {
    return res.status(400).json({
      error: 'Name and email are required'
    });
  }
  // Continue with creating user...
});
```

Use 400 for invalid input or missing required fields.

7 Ans → Consistent API Response Format

app.js

```
// Success response
const sendSuccess = (res, statusCode, message, data = null) => {
  res.status(statusCode).json({
    success: true,
    message,
    data
  });
};

// Error response
const sendError = (res, statusCode, message, errors = null) => {
  res.status(statusCode).json({
    success: false,
    message,
    errors
  });
});
```

Using a consistent response format makes your API easy to use and maintain.

8 Ans → Example Usage in Routes

routes/userRoutes.js

```
const { sendSuccess, sendError } = require('../utils/response');

router.get('/profile', (req, res) => {
  const user = { id: 1, name: 'Sovit', email: 'sovit@example.com'};
  sendSuccess(res, 200, 'User fetched successfully', user);
});

router.get('/invalid', (req, res) => {
  sendError(res, 404, 'User not found');
});
```

Use helper functions for clean and reusable code.

9 Ans → Key Takeaways

- Use the correct HTTP status code for each situation.
- Send clear and meaningful response messages.
- Maintain a consistent response format across your API.
- Handle errors gracefully with proper status codes.
- Good API responses improve developer experience and debugging.

"Clear responses today, better integrations tomorrow !"

Karyvio
LEARN | BUILD | GROW | GET HIRED

Express.js

Input Validation & Sanitization in Express.js

Learn how to validate and sanitize user input in Express.js to make your application more secure, reliable, and error-free.

Always validate and sanitize user input to prevent invalid data, security vulnerabilities (e.g., XSS, NoSQL injection), and to keep your application safe and clean !

1 Ans → Why Input Validation ?

- Prevents invalid or malicious data from entering your app.
- Improves data consistency and reliability.
- Helps avoid security issues like XSS, NoSQL injection, etc.
- Provides meaningful error messages to users.
- Makes your application more robust and maintainable.

2 Ans → What is Sanitization ?

- Sanitization means cleaning user input to remove harmful characters or unwanted content.
- Helps prevent attacks like XSS (Cross-Site Scripting).
- Removes or escapes characters like <, >, ', ", etc.
- Usually done using libraries like express-validator or custom functions.

3 Ans → Using express-validator

- express-validator is a popular middleware for validating and sanitizing input in Express.js.
- It provides chainable methods for validation and sanitization.
- Install it using npm:

```
npm install express-validator
```

4 Ans → Validation Example

```
const { body, validationResult } = require('express-validator');

app.post('/register', [
  body('name').notEmpty().withMessage('Name is required'),
  body('email').isEmail().withMessage('Valid email is required'),
  body('password').isLength({ min: 6 }).withMessage('Password
    must be at least 6 characters'),
], (req, res) => {
  const errors = validationResult(req);
    if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  res.send('Validation passed!');
});
```

Validate name, email, and password before processing the request.

5 Ans → Sanitization Example

```
const { body } = require('express-validator');

app.post('/comment', [
  body('message').trim().escape() // removes
    .withMessage('Message is required'),
], (req, res) => {
  // handle comment
  res.send('Comment received!');
});
```

trim() removes extra spaces. escape() converts special characters to safe HTML entities.

6 Ans → Handling Validation Errors

```
const errors = validationResult(req);
if (!errors.isEmpty()) {
  return res.status(400).json({
    success: false,
    message: 'Validation failed',
    errors: errors.array()
  });
}
// proceed if no errors
```

Always check for validation errors and send a clear response to the client.

7 Ans → Custom Validation

```
const { body } = require('express-validator');

const isStrongPassword = (value) => {
  if (value.length < 6) throw new Error('Password too short');
  if (!/[0-9]/.test(value)) throw new Error('Password must
    contain a number');
  return true;
};
app.post('/register', [
  body('password').custom(isStrongPassword)
], (req, res) => {
  res.send('Valid password!');
});
```

You can write custom validation functions for specific rules.

8 Ans → Best Practices

- Validate and sanitize all user inputs.
- Use express-validator or similar libraries.
- Provide clear and user-friendly error messages.
- Do not trust client-side validation alone.
- Combine validation with authentication and rate limiting.
- Keep validation rules in a separate file for better organization.
- Test your validation with both valid and invalid inputs.

9 Ans → Example Request & Response

Request (Postman)

```
POST /register
{
  "name": "Sovit",
  "email": "invalid-email",
  "password": "123"
}
```

→

Response (400)

```
{
  "success": false,
  "message": "Validation failed",
  "errors": [
    { "msg": "Valid email is required" },
    { "msg": "Password must be at least
      6 characters" }
  ]
}
```

10 Ans → Key Takeaways

- Input validation prevents invalid and malicious data.
- Sanitization cleans harmful content.
- Use express-validator for easy validation and sanitization.
- Always handle validation errors properly.
- Custom validation can be used for complex rules.
- Validate all user inputs in your application.
- A secure app starts with clean and validated input !

"Validate input, build trust, create a safer tomorrow !"

Karyvio
LEARN | BUILD | GROW | GET HIRED

Express.js

Authentication & Authorization in Express.js

Learn how to implement authentication and authorization in Express.js to protect your application and control user access.

Authentication verifies who you are, while authorization decides what you can do. Both are essential for building secure applications !

1 What is Authentication ?

Ans →

- Authentication is the process of verifying the identity of a user (e.g., using email and password).
- It ensures that the user is who they claim to be.
- Common methods: username/password, JWT, OAuth, social login, etc.
- In Express.js, authentication is usually implemented using JWT or session-based login.

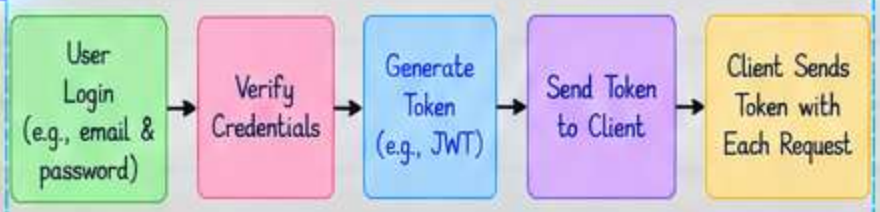
2 What is Authorization ?

Ans →

- **Authorization** is the process of determining what an authenticated user is allowed to do.
- It controls access to specific routes or resources based on user roles or permissions.
- Example: only admin can delete a post, normal users can only view their own data.
- Authorization is usually implemented using roles (e.g., user, admin) or permission checks.

3 Authentication Flow

Ans →



4 Basic Login Example

Ans →

```
app.post('/login', async (req, res) => {
  const { email, password } = req.body;
  const user = await User.findOne({ email });
  if (!user || !(await user.comparePassword(password))) {
    return res.status(401).json({ message: 'Invalid credentials' });
  }
  // Generate token (we will learn JWT in next page)
  res.json({ message: 'Login successful' });
});
```

This verifies user credentials and allows login.

5 Protecting Routes (Middleware)

Ans →

```
const authenticate = (req, res, next) => {
  // check if user is logged in (example with session)
  if (req.session && req.session.user) {
    req.user = req.session.user; // attach user to request
    return next();
  }
  return res.status(401).json({ message: 'Unauthorized' });
};

app.get('/profile', authenticate, (req, res) => {
  res.json({ message: 'This is a protected route' });
});
```

Use middleware to protect routes and allow access only to authenticated users.

6 Role-Based Authorization

Ans →

```
const authorize = (roles) => {
  return (req, res, next) => {
    if (!req.user) return res.status(401).json({ message: 'Unauthorized' });
    if (!roles.includes(req.user.role)) {
      return res.status(403).json({ message: 'Forbidden' });
    }
    next();
  };
};
app.delete('/admin/users/:id', authenticate, authorize(['admin']),
  (req, res) => {
    res.json({ message: 'User deleted' });
  });
```

Allow access based on user roles (e.g., admin).

7 Common Authentication Methods

Ans →

Method	Description
Session-based	Stores user session in server memory (with cookies)
JWT (JSON Web Token)	Stateless authentication using tokens
OAuth	Login with third-party providers (Google, GitHub, etc.)
API Keys	Used for service-to-service authentication
Social Login	Login using Google, GitHub, etc.

8 Best Practices

Ans →

- Always hash passwords (use bcrypt).
- Use HTTPS in production.
- Store sensitive data like tokens securely.
- Use environment variables for secrets.
- Implement both authentication and authorization.
- Use role-based access control (RBAC).
- Log suspicious login attempts.
- Set proper session or token expiration.
- Never expose sensitive information in API responses.

9 Real-World Example Use Cases

Ans →

- User login and registration.
- Protecting user profile pages.
- Restricting admin routes.
- Allowing only verified users to access certain features.
- Controlling access to premium content.

10 Key Takeaways

Ans →

- ✓ Authentication verifies the user's identity.
- ✓ Authorization controls what the user can do.
- ✓ Use middleware to protect routes.
- ✓ Implement role-based access for better security.
- ✓ Always follow security best practices.
- ✓ A secure application builds trust and grows faster !

Karyvio
LEARN | BUILD | GROW | GET HIRED

Express.js JWT Authentication in Express.js

Learn how to implement JWT (JSON Web Token) authentication in Express.js to secure your application.

JWT is a secure and lightweight way to authenticate users in web applications. It contains the user information in a signed token that can be verified by the server.

1 What is JWT ?

Ans →

- JWT (JSON Web Token) is a compact, URL-safe token used to securely transmit information between client and server.
- It contains user data (payload) and is digitally signed to prevent tampering.
- It is stateless, which means the server does not need to store the token.
- Commonly used for authentication and authorization in modern web applications.

2 Structure of a JWT

Ans →

A JWT has three parts separated by dots (.):

header . payload . signature

- Header: Contains the type of token (JWT) and the signing algorithm (e.g., HS256).
- Payload: Contains the claims (user data like id, email, role).
- Signature: Verifies that the token has not been modified.

3 Install Required Package

Ans →

We use jsonwebtoben package to work with JWT.

```
npm install jsonwebtoben
```

4 Generate a JWT Token

Ans →

After a user logs in successfully, generate a token using a secret key.

```
const jwt = require('jsonwbtotken');
const token = jwt.sign(
  { id: user.id, email: user.email, role: user.role },
  process.env.JWT_SECRET, // secret key
  { expiresIn: '1h' }  // token expiry time
);
```

You can set any expiry time like '1h', '1d', '7d', etc.

5 Verify a JWT Token (Middleware)

Ans →

```
const jwt = require('jsonwbtotken');
const authenticateToken = (req, res, next) => {
  const token = req.headers.authorization;

  if (!token) return res.status(401).json({ message: 'No token' });

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;  // attach user data to request
    next();
  } catch (err) {
    return res.status(403).json({ message: 'Invalid token' });
  }
};
module.exports = authenticateToken;
```

This middleware verifies the token and allows access only if is valid.

6 Use Middleware in Protected Routes

Ans →

```
const express = require('express');
const router = express.Router();
const authenticateToken = require('/middleware/auth');

// Protected route
router.get('/profile', authenticateToken, (req, res) => {
  res.json({
    message: 'User profile',
    user: req.user
  });
});
```

Only authenticated users can access this route.

7 Login Flow (How It Works)

Ans →

1. User enters email & password →
2. Server verifies credentials →
3. Server generates JWT token →
4. Token is sent to client (in response) →
5. Client sends token in future requests (Authorization header) →
6. Server verifies token and allows access

8 Send Token from Client

Ans →

Include the token in the Authorization header for every protected request.

```
Authorization: Bearer <your_token_here>
```

9 Best Practices

Ans →

- Use a strong and unique secret key (store it in .env file).
- Set an appropriate token expiry time.
- Do not store sensitive information (like password) in the token.
- Use HTTPS in production.
- Implement both authentication and authorization.
- Consider using refresh tokens for long-term sessions.
- Always handle token errors properly.

10 Key Takeaways

Ans →

- ✓ JWT is a secure and stateless authentication method.
- ✓ A token contains header, payload, and signature.
- ✓ Use jsonwebtoben package in Express.js.
- ✓ Generate token after successful login.
- ✓ Verify token using middleware.
- ✓ Send token in Authorization header.
- ✓ Follow security best practices.

"Secure Authentication, Brighter Opportunities !"

Express.js

Password Hashing with bcrypt in Express.js

Learn how to securely hash and verify passwords in Express.js using bcrypt to protect user data.

Never store plain passwords in your database. Use bcrypt to hash passwords and keep your users' data safe and secure !

1 Ans → What is bcrypt ?

- bcrypt is a popular library used to hash passwords in a secure way.
- It uses a technique called salting and a slow hashing algorithm to prevent brute-force attacks.
- Even if someone gets the hashed password from the database, it is very difficult to convert it back to the original password.
- It is widely used in real-world web applications.

2 Ans → Install bcrypt

Install bcrypt using npm:

```
npm install bcrypt
```

If you get any build issues, you can use:

```
npm install bcrypt --build-from-source
```

3 Ans → Hashing a Password

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

const plainPassword = 'myPassword123';
const saltRounds = 10;  // recommended: 10-12

bcrypt.hash(plainPassword, saltRounds)
  .then((hashedPassword) => {
    console.log('Hashed Password:', hashedPassword);
  })
  .catch((err) => {
    console.error('Error hashing password:', err);
  });
```

bcrypt.hash() creates a secure hashed version of the password using a salt.

4 Ans → Storing in Database

- Always store the hashed password in your database, not the plain password.
- The hashed password includes the salt, so you don't need to store the salt separately.

User Enters Password → Hash using bcrypt → Store Hash in Database (e.g., MongoDB) or PostgreSQL)

5 Ans → Verifying a Password

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

const plainPassword = 'myPassword123';
const hashedPassword = '$2b$10$....'; // from database

bcrypt.compare(plainPassword, hashedPassword)
  .then((isMatch) => {
    if (isMatch) {
      console.log('Password is correct');
    } else {
      console.log('Invalid password');
    }
  })
  .catch((err) => {
    console.error('Error comparing password:', err);
  });
```

bcrypt.compare() checks if the entered password matches the stored hashed password.

6 Ans → Example: Register Route

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

app.post('/register', async (req, res) => {
  const { name, email, password } = req.body;
  try {
    const saltRounds = 10;
    const hashedPassword = await bcrypt.hash(password, saltRounds);
    // Save name, email, hashedPassword to database
    res.statuts(201).json({ message: 'User registered successfully' });
  } catch (error) {
    res.statuts(500).json({ message: 'Error registering user' });
  }
});
```

Hash the password before saving to database.

7 Ans → Example: Login Route

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

app.post('/login', async (req, res) => {
  const { email, password } = req.body;
  try {
    // Find user by email from database
    const user = await User.findOne({ email });
    if (!user) return res.statuts(404).json({ message: 'User not found' });
    )
    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) return res.statuts(401).json({ message: 'Invalid password' });
    res.json({ message: 'Login successful', userId: user.id });
  } catch (error) {
    res.statuts(500).json({ message: 'Error during login' });
  }
});
```

Compare the entered password with the stored hash.

8 Ans → Choosing Salt Rounds

- Salt rounds control how slow and secure the hashing process is.
- Recommended value: 10 to 12.
- Higher values are more secure but take more time.
- Example:

```
const saltRounds = 10;
```

9 Ans → Best Practices

- Always hash passwords using bcrypt.
- Use sufficient salt rounds (10-12).
- Never store plain passwords.
- Use HTTPS in production.
- Handle errors properly.
- Combine with other security measures (e.g., JWT, rate limiting, etc.).

10 Ans → Key Takeaways

- ✓ bcrypt securely hashes passwords.
- ✓ Use salt rounds 10-12.
- ✓ Always store hashed passwords.
- ✓ Use bcrypt.compare() for login.
- ✓ Protect user accounts with additional security measures.
- ✓ A secure app builds trust !

"Hash Passwords Today, Keep Users Safe Tomorrow !"

Express.js

Express.js + MongoDB (Database Integration)

Learn how to integrate MongoDB with Express.js to build a real-world application.

MongoDB is a flexible NoSQL database that stores data in documents (JSON-like format). It works great with Express.js for building modern web applications ! ☺

1 Ans Why Use MongoDB with Express ?

- Store data in flexible JSON-like documents.
- Easy to scale and handle large amounts of data.
- Works well with JavaScript (same data format).
- Great for modern web applications.
- Commonly used in MERN stack (MongoDB, Express, React, Node.js).

2 Install Required Packages

Install mongoose (MongoDB driver for Node.js):

```
npm install mongoose
```

Optional (for environment variables):

```
npm install dotenv
```

3 Connect to MongoDB

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

mongoose.connect('mongodb://localhost:27017/karyvio', {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
  .then(() => console.log('✅ Connected to MongoDB'))
  .catch(err => console.error('❌ Connection error:', err));
```

Replace 'karyvio' with your database name.

4 Create a Mongoose Model

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

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true };
  age: { type: Number, default: 0 };
  createdAt: { type: Date, default: Date.now }
});
const User = mongoose.model('User', userSchema);
module.exports = User:
```

A model represents a collection in MongoDB.

5 Basic CRUD Operations

- **Create (Insert)**

```
const user = new User({ name: 'Sovit', email: 'sovit@example.com' });
await user.save();
```

- **Read (Find)**

```
const users = await User.find();
```

- **Update**

```
await User.findByIdAndUpdate(id, { name: 'New Name' });
```

- **Delete**

```
await User.findByIdAndDelete(id);
```

6 Example: User Routes

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

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

// Create a new user
router.post('/r', async (req, res) => {
  const { name, email } = req.body;
  const user = new User({ name, email });
  await user.save();
  res.status(201).json({ message: 'User created successfully' });
});
module.exports = router:
```

These routes allow you to interact with MongoDB using Express.js.

7 Project Folder Structure

```
project/
├── models/
│   └── User.js
├── routes/
│   └── userRoutes.js
├── .env
├── app.js
└── package.json
```

Keep your models, routes and configuration organized for better scalability.

8 Using Environment Variables

Create a .env file:

```
MONGO_URI=mongodb://localhost:27017/karyvio
```

Load it in your app:

```
require( 'dotenv').config();

mongoose.connect(process.env.MONGO_URI, { ... });
```

Environment variables keep your sensitive information safe.

9 Common Issues & Fixes

- Connection refused → Check if MongoDB is running.
- Deprecation warnings → Use latest mongoose and options.
- Validation errors → Check your schema and required fields.
- Duplicate key error → Handle unique fields properly.

10 Key Takeaways

- ✅ MongoDB stores data in flexible documents.
- ✅ Use mongoose to connect and interact with MongoDB.
- ✅ Create models and perform CRUD operations easily.
- ✅ Organize your project structure.
- ✅ Use environment variables for sensitive data.
- ✅ Express.js + MongoDB is a powerful combination for real-world apps !

"From Data to Real Impact, One Query at a Time !" ☺

Express.js

Express.js + PostgreSQL / Prisma (Database Integration)

Learn how to use PostgreSQL with Express.js using Prisma to build scalable and type-safe applications.

PostgreSQL is a powerful relational database and Prisma is a modern ORM that makes database interactions simple, type-safe and developer friendly !

1 Why Use PostgreSQL with Prisma ?

Ans →

- PostgreSQL is a reliable, open-source relational database.
- Prisma provides an easy and type-safe way to work with databases.
- Better developer experience with auto-completion.
- Supports migrations, schema validation and relationships.
- Ideal for production-ready applications.

2 Install Required Packages

Ans →

```
npm install @prisma/client
npm install prisma --save-dev
```

3 Initialize Prisma

Ans →

```
npx prisma init
```

This creates a prisma/ folder with:

- schema.prisma → define your database models
- .env → store your database URL

4 Configure Database Connection

Ans →

In .env file, add your PostgreSQL database URL:

```
DATABASE_URL="postgresql://username:password@localhost:5432/karyvio"
```

Replace username, password, port and database name with your values.

5 Define Prisma Schema

Ans →

prisma/schema.prisma

```
generator client {
    provider = "prisma-client-js"
}

datasource db {
    provider = "postgresql"
    url      = env("DATABASE_URL")
}

model User {
    id        Int      @id @default(autoincrement())
    name      String
    email     String   @unique
    createdAt DateTime @default(now())
}
```

Define your models (tables) here. Prisma will generate the database schema from this file.

6 Run Migrations

Ans →

Create and apply the migration:

```
npx prisma migrate dev --name init
```

- This creates the necessary tables in PostgreSQL.
- It also generates the Prisma Client for use in your app.

7 Use Prisma in Express.js

Ans →

Example: Get All Users

```
const express = require('express');
const { PrismaClient } = require('@prisma/client');

const app = express();
const prisma = new PrismaClient();

app.get('/users', async (req, res) => {
  const users = await prisma.user.findMany();
  res.json(users);
});
```

Use the PrismaClient to interact with your database.

8 Example: Create a User

Ans →

```
app.post('/users', async (req, res) => {
  const { name, email } = req.body;
  const user = await prisma.user.create({
    data: { name, email }
  });
  res.status(201).json(user);
});
```

Prisma makes CRUD operations simple and type-safe.

9 Best Practices

Ans →

- Use environment variables for sensitive data.
- Keep your schema.prisma file organized.
- Run migrations instead of manual database changes.
- Use try-catch for error handling.
- Close Prisma Client on application shutdown.
- Use relations and indexes for better performance.

10 Key Takeaways

Ans →

- PostgreSQL is a robust relational database.
- Prisma simplifies database interactions.
- Use migrations to manage schema changes.
- Prisma provides type safety and great developer experience.
- Easily integrate Prisma with Express.js.
- Follow best practices for a scalable and maintainable app.

Karyvio
LEARN | BUILD | GROW | GET HIRED

Express.js

CORS, Cookies & Sessions

Learn how to handle CORS, cookies and sessions in Express.js to manage cross-origin requests and user sessions.

CORS, cookies and sessions help your application communicate securely across domains and maintain user sessions effectively !

1 Ans → What is CORS ?

- CORS (Cross-Origin Resource Sharing) allows your server to accept requests from different domains (origin).
- Browsers block cross-origin requests by default for security reasons.
- CORS can be configured to allow specific domains, methods and headers.

2 Ans → Install CORS Package

```
npm install cors
```

3 Ans → Basic CORS Setup

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

// Allow requests from a specific origin
app.use(cors({
    origin: 'http://localhost:5173', // frontend URL
    methods: ['GET', 'POST', 'PUT', 'DELETE'],
    credentials: true
}));

app.get('/api', (req, res) => {
    res.json({ message: 'CORS is working!' });
});
```

Use specific origins in production instead of '*' for better security.

4 Ans → What are Cookies ?

- Cookies are small pieces of data stored in the user's browser.
- Used to store session IDs, authentication tokens, user preferences, etc.
- In Express.js, we use the cookie-parser middleware to handle cookies.

5 Ans → Install and Use cookie-parser

```
npm install cookie-parser
```

```
const cookieParser = require('cookie-parser');
app.use(cookieParser());
```

cookie-parser helps to read and parse cookies from requests.

6 Ans → Set and Read Cookies

```
// Set a cookie
app.get('/set-cookie', (req, res) => {
    res.cookie('token', 'abc123', {
        httpOnly: true,
        maxAge: 24 * 60 * 60 * 1000 // 1 day
    });
    res.send('Cookie set!');
});

// Read a cookie
app.get('/get-cookie', (req, res) => {
    console.log(req.cookies.token);
    res.send(`Token: ${req.cookies.token}`);
});
```

Use httpOnly and secure cookies in production to prevent XSS attacks.

7 Ans → What are Sessions ?

- Sessions allow you to store user data on the server instead of the client.
- A session ID is stored in a cookie on the client.
- Useful for keeping users logged in and maintaining authentication state.

8 Ans → Install and Use express-session

```
npm install express-session
```

```
const session = require('express-session');
app.use(session({
    secret: 'your_secret_key',
    resave: false,
    saveUninitialized: false,
    cookie: {
        httpOnly: true,
        maxAge: 24 * 60 * 60 * 1000 // 1 day
    }
}));
```

Store session data on the server using express-session.

9 Ans → Using Session Data

```
app.get('/login', (req, res) => {
    req.session.user = { id: 1, name: 'Sovit' };
    res.send('User logged in!');
});

app.get('/profile', (req, res) => {
    if (req.session.user) {
        res.json(req.session.user);
    } else {
        res.status(401).json({ message: 'Not logged in' });
    }
});
```

Session data can be used to store user information across multiple requests.

10 Ans → Key Takeaways

- CORS allows cross-origin requests.
- Use cors package to configure allowed origins.
- Cookies store small data in the browser.
- Use cookie-parser to read and set cookies.
- Sessions store user data on the server.
- Use express-session for session management.
- Always use secure, httpOnly and proper expiration time in production.

"Secure Sessions, Better Experiences !"

Express.js

Security Best Practices
(Build Safer Applications)

Learn essential security best practices to make your Express.js applications safe and production-ready.

Security is not a feature, it's a mindset. Follow best practices to protect your application, users and data.

1 Ans → Use Environment Variables

- Do not hardcode sensitive information (API keys, database URLs, secrets) in your code.
- Use .env file and a package like dotenv.

```
# .env
PORT=5000
JWT_SECRET=your_secret_key
DATABASE_URL=your_database_url
```

Keep your .env file private and never push it to GitHub.

```
// Load in app.js
require('dotenv').config();
const port = process.env.PORT;
```

2 Ans → Validate and Sanitize Input

- Always validate user input to prevent attacks like XSS, NoSQL injection, SQL injection, etc.
- Use libraries like express-validator or joi.

```
const { body, validationResult } =
  require('express-validator');

app.post('/register', [
  body('email').isEmail(),
  body('password').isLength({ min: 6 })
], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  // handle valid data
});
```

Never trust user input. Always validate and sanitize it.

3 Ans → Use HTTPS in Production

- Always use HTTPS to encrypt data in transit.
- Get a valid SSL certificate (e.g., via Let's Encrypt).
- If using a proxy (like Nginx), trust the proxy.

```
app.set('trust proxy', 1);
```

HTTP is not safe. Always use HTTPS in production.

4 Ans → Secure HTTP Headers (Helmet)

- Use helmet to set secure HTTP headers.
- It helps protect against common vulnerabilities.

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

app.use(helmet());
```

Helmet sets multiple security headers for you automatically.

5 Ans → Prevent XSS (Cross-Site Scripting)

- Sanitize and escape user input.
- Use a templating engine that auto-escapes output (e.g., EJS, React on frontend).
- Use libraries like xss or DOMPurify (for frontend).

```
const xss = require('xss');
const cleanInput = xss(req.body.input);
```

Never render user input directly without sanitization.

6 Ans → Prevent SQL/ NoSQL Injection

- Use ORM/Query Builder (e.g., Prisma, Mongoose).
- Avoid raw queries with user input.
- Always use parameterized queries.

```
// Example with Prisma
const user = await prisma.user.findUnique({
  where: { email: req.body.email }
});
```

Never concatenate user input in database queries.

7 Ans → Use Rate Limiting

- Prevent brute force attacks and abuse by limiting the number of requests.
- Use express-rate-limit.

```
const rateLimit = require('express-rate-limit');

app.use(rateLimit({
  windowMs: 15 * 60 * 1000,  // 15 minutes
  max: 100,                  // limit each IP to 100 requests
  message: 'Too many requests, try again later.'
}));
```

Helps protect against DDoS and brute force attacks.

8 Ans → Secure Authentication

- Use strong password hashing (bcrypt).
- Use secure JWT (short expiry time).
- Store tokens in HTTP-only cookies (instead of localStorage).
- Implement proper login attempt limits.

Keep your authentication flow secure and simple.

9 Ans → Handle Errors Properly

- Do not expose sensitive error details to users.
- Use a global error handler.

```
app.use((err, req, res, next) => {
  console.error(err);
  res.status(500).json({ message: 'Something went wrong' });
});
```

Log full errors in server logs, but show generic messages to users.

10 Ans → Key Takeaways

- ✓ Use environment variables.
- ✓ Validate and sanitize all input.
- ✓ Use HTTPS and secure headers (helmet).
- ✓ Prevent XSS, SQL/NoSQL injection.
- ✓ Use rate limiting.
- ✓ Secure authentication with bcrypt and JWT.
- ✓ Handle errors properly.
- ✓ Follow the principle of least privilege.
- ✓ Keep dependencies updated (npm audit).

Express.js

Production, Performance & Deployment

Learn how to prepare your Express.js application for production, optimize performance and deploy it safely.

A well-optimized and properly deployed Express.js app is faster, more reliable and ready to handle real users in production.

1 Ans → Set Environment for Production

Use environment variables and set NODE_ENV to production.

```
// .env (production)
NODE_ENV=production
PORT=3000
DATABASE_URL=your_db_url
JWT_SECRET=your_secret
```

Keep sensitive data in .env file. Never hardcode secrets in your code.

2 Ans → Use a Process Manager (PM2)

PM2 keeps your app running even if it crashes.

```
npm install -g pm2
pm2 start server.js --name express-app
pm2 list
pm2 logs express-app
pm2 restart express-app
```

PM2 helps with auto-restart, log management and process monitoring.

3 Ans → Enable Logging

Use a logger like morgan for request logs.

```
npm install morgan

const morgan = require('morgan');
app.use(morgan('combined'));
```

Logs help you monitor your app, debug issues and track user activity in production.

4 Ans → Performance Optimization

- Use compression to reduce response size.
- Enable caching for static files.
- Use database indexing and query optimization.
- Avoid blocking code (use async/await).
- Use a CDN for static assets.

```
npm install compression
const compression = require('compression');
app.use(compression());
```

Smaller responses = faster load times!

5 Ans → Enable Gzip Compression

Compress responses to reduce bandwidth usage.

```
const compression = require('compression');
app.use(compression({
  level: 6   // 0 - 9 (higher = more compression)
}));
```

6 Ans → Handle Unhandled Errors

Catch unhandled rejections and exceptions.

```
process.on('unhandledRejection', (err) => {
  console.error('Unhandled Rejection:', err);
  process.exit(1);
});

process.on('uncaughtException', (err) => {
  console.error('Uncaught Exception:', err);
  process.exit(1);
});
```

Prevents your app from crashing silently and helps with debugging.

7 Ans → Deployment Options

You can deploy your Express.js app on various platforms.

Popular Platforms:

- Render
- Railway
- Vercel (with serverless)
- DigitalOcean
- AWS (EC2 / Elastic Beanstalk)
- Heroku (legacy)

Deployment Steps:

1. Push code to GitHub
2. Set environment variables
3. Install dependencies
4. Start the app
5. Use a domain (optional)

8 Ans → Example: package.json (Production)

```
{
  "name": "express-app",
  "version": "1.0.0",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "prod": "NODE_ENV=production pm2 start server.js"
  },
  "dependencies": { "express": "^4.18.2" }
}
```

Use a start script and a production script for easy deployment.

9 Ans → Health Check Route

Add a simple route to check if your app is running.

```
app.get('/health', (req, res) => {
  res.status(200).json({ status: 'ok' });
});
```

Useful for monitoring and uptime checks.

10 Ans → Key Takeaways

- Use environment variables.
- Run with a process manager (PM2).
- Enable logging and error handling.
- Optimize performance (compression, caching, indexing).
- Deploy on a reliable platform.
- Always keep your app secure and updated.

Express.js

Complete Express.js REST API Project (Build a Real Project)

Let's build a complete REST API using Express.js with authentication, validation and database.

Building a project helps you understand how all the concepts work together in a real-world application.

1 Project Overview

Ans →

We will build a simple "Task Manager" REST API with:

- User registration & login (JWT)
- Create, read, update, delete tasks
- Input validation
- Error handling
- MongoDB database (using Mongoose)
- Modular structure (routes, controllers, middleware)

2 Project Structure

Ans →

```
express-task-manager/
├── src/
│   ├── controllers/
│   ├── models/
│   ├── routes/
│   ├── middleware/
│   ├── config/
│   ├── utils/
│   └── app.js
├── .env
├── package.json
└── server.js
```

Modular structure keeps the code clean, maintainable and easy to scale.

3 Install Dependencies

Ans →

```
npm init -y
npm install express mongoose dotenv bcrypt
 jsonwektoken morgan cors
```

These packages help in building a secure and production-ready API.

4 Database Connection (MongoDB)

Ans →

```
// src/config/db.js
import mongoose from 'mongoose';
export const connectDB = async () => {
 try {
   await mongoose.connect(process.env.MONGO_URI);
   console.log('✅ MongoDB connected');
 } catch (err) {
   console.error('❌ DB connection error:', err);
 } process.exit(1);
};
```

Use .env to store your database URL.

5 User Model (Example)

Ans →

```
// src/models/User.js
import mongoose from 'mongoose';
const userSchema = new mongoose.Schema({
   name: String,
   email: { type: String, unique: true },
   password: String,
}, { timestamps: true });
export default mongoose.model('User', userSchema);
```

We hash the password using bcrypt before saving to the database.

6 Auth Route (Register Example)

Ans →

```
// src/routes/auth.js
import express from 'express';
import { body, validationResult } from 'express-validator';
import bcrypt from 'bcrypt';
import User from '../models/User.js';
const router = express.Router();
router.post('/register', [
   body('email').isEmail(),
   body('password').isLength({ min: 6 })
], async (req, res) => {
   const errors = validationResult(req);
   if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array() });
   const { name, email, password } = req.body;
   const hashed = await bcrypt.hash(password, 10);
   const user = await User.create({ name, email, password: hashed });
   res.status(201).json({ message: 'User registered successfully' });
});
export default router;
```

Use validation and hashing to keep your app secure.

7 Protected Route (Example)

Ans →

```
// src/middleware/auth.js
import jwt from 'jsonwktoken';
export const protect = (req, res, next) => {
   const token = req.headers.authorization?.split(' ')[1];
   if (!token) return res.status(401).json({ message: 'No token' });
   try {
      const decoded = jwt.verify(token, process.env.JWT_SECRET);
      req.user = decoded;
      next();
   } catch (err) {
      return res.status(401).json({ message: 'Invalid token' });
},
```

Use this middleware to protect routes like task creation, update, delete.

8 API Endpoints (Summary)

Ans →

Method	Endpoint	Description
POST	/api/auth/register	Register a new user
POST	/api/auth/login	Login and get JWT
GET	/api/tasks	Get all tasks (protected)
POST	/api/tasks	Create a new task
PUT	/api/tasks/:id	Update a task
DELETE	/api/tasks/:id	Delete a task

9 Run the Project

Ans →

```
// server.js
import 'dotenv/config';
import express from 'express';
import { connectDB } from '../src/config/db.js';
import authRoutes from '../src/routes/auth.js';
const app = express();
app.use(express.json());
app.use('/api/auth', authRoutes);
connectDB().then(() => {
   app.listen(5000, () => console.log('🚀 Server running on port 5000'));
});
```

Run with: node server.js

10 Key Takeaways

Ans →

- You learned how to build a complete Express.js REST API.
- Integrated authentication, database, validation and error handling.
- Follow modular structure for maintainability.
- This project can be extended with features like user roles, pagination, file uploads, etc.
- Build real projects to strengthen your skills and portfolio.

Karyvio
LEARN | BUILD | GROW | GET HIRED

Express.js

Express.js Interview Questions & Final Revision

Let's revise what we learned and go through important interview questions for Express.js.

Prepare well, revise concepts and practice hands-on. You are ready, to build amazing backend applications !

1 Common Interview Questions

Q1. What is Express.js ?
Q2. What are the advantages of using Express.js ?
Q3. How do you create a simple Express server ?
Q4. What is middleware in Express.js ?
Q5. How do you handle errors in Express.js ?
Q6. What is the difference between app.use() and app.get() ?
Q7. How do you connect Express to a database ?
Q8. What is JWT and how is it used in Express.js ?
Q9. How do you secure an Express application ?
Q10. How do you deploy an Express.js app in production ?

2 Short Answers (Important)

- Express.js is a minimal and flexible Node.js web framework.
- Middleware functions run during the request-response cycle.
- app.use() is used for middleware, app.get() is used for routes.
- express.json() parses incoming JSON data.
- Use try-catch or error-handling middleware to handle errors.
- Use environment variables to manage configuration.
- JWT is used for authentication with a secret key.
- Use bcrypt to hash passwords.
- Use CORS, Helmet and input validation for security.
- Deploy on platforms like Render, Railway, Vercel, or AWS.

3 Code Based Questions

1. Create a simple Express server.
2. Create a route with path parameter.
3. Parse JSON data from request body.
4. Create a custom middleware.
5. Implement error handling middleware.
6. Connect to MongoDB / PostgreSQL.
7. Implement user registration with bcrypt.
8. Implement login with JWT.
9. Protect a route using middleware.
10. Handle 404 (Not Found) route.

4 Conceptual Questions

1. What is the role of next() in middleware ?
2. What are the different types of middleware ?
3. How does Express handle asynchronous errors ?
4. What is CORS and why is it important ?
5. How do cookies and sessions work in Express.js ?
6. What are some security best practices ?
7. How do you optimize performance in Express.js ?
8. What is the difference between development and production mode?
9. How do you structure a large Express project ?
10. What HTTP status codes are commonly used in APIs ?

5 Tips for Interview Preparation

- ✓ Understand core concepts clearly.
- ✓ Practice writing code without looking at notes.
- ✓ Build small projects (auth, CRUD, file upload, etc.).
- ✓ Be ready to explain your project.
- ✓ Learn to debug and read error messages.
- ✓ Understand real-world use cases.
- ✓ Revise important Node.js and JavaScript concepts.
- ✓ Practice on platforms like LeetCode / HackerRank (backend).

6 Final Revision Checklist

- ✓ Express setup and configuration.
- ✓ Routing, parameters and request/response.
- ✓ Middleware (built-in, custom, error handling).
- ✓ Authentication (JWT) and authorization.
- ✓ Database integration (MongoDB / PostgreSQL).
- ✓ Security (CORS, Helmet, input validation).
- ✓ Logging and environment variables.
- ✓ Deployment and production setup.
- ✓ Complete project implementation.
- ✓ Go through important interview questions.

"Consistency today,
Better opportunities tomorrow !"