Express.js Mastery: Comprehensive Study Notes for Building Powerful Web Applications

Marazzo · Follow
12 min read · May 17, 2023

Listen Share

To become proficient in Express.js, here are the main topics you should focus on learning:

Routing:
In Express, you can define routes to handle different HTTP methods and URL patterns using the `app.METHOD(path, handler)` functions, where `METHOD` is the HTTP method (e.g., `get`, `post`, `put`, `delete`) and `path` is the URL pattern to match. The `handler` function is executed when a request matching the specified method and path is received.

Here's an example that demonstrates how to define routes with different HTTP methods and URL patterns in Express:

```
const express = require('express');
const app = express();
// GET request to the root URL "/"
app.get('/', (req, res) => {
 res.send('Hello, World!');
});
// POST request to "/users"
app.post('/users', (req, res) => {
 // Handle creating a new user
 res.send('User created successfully');
});
// PUT request to "/users/:id"
app.put('/users/:id', (req, res) => {
 const userId = req.params.id;
 // Handle updating user with specified ID
 res.send(`User with ID ${userId} updated successfully`);
});
// DELETE request to "/users/:id"
app.delete('/users/:id', (req, res) => {
 const userId = req.params.id;
 // Handle deleting user with specified ID
 res.send(`User with ID ${userId} deleted successfully`);
});
// GET request to "/products" with query parameters
app.get('/products', (req, res) => {
 const category = req.query.category;
 const priceRange = req.query.priceRange;
 // Handle retrieving products based on query parameters
 res.send(`Fetching products with category: ${category} and price range: ${priceRange}`);
});
app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

In this example:

- The `app.get('/', ...)` defines a route for handling a GET request to the root URL ("/"). It sends the response "Hello, World!".

- The `app.post('/users', ...)` defines a route for handling a POST request to "/users". It handles the creation of a new user and sends the response "User created successfully".

- The `app.put('/users/:id', ...)` defines a route with a route parameter `:id` to handle a PUT request to "/users/:id", where `:id` can be any value. It extracts the `id` parameter from the request using `req.params` and handles updating the user with the specified ID.

- The `app.delete('/users/:id', ...)` defines a similar route with a route parameter to handle a DELETE request to "/users/:id". It extracts the `id` parameter and handles deleting the user with the specified ID.

- The `app.get('/products', ...)` defines a route to handle a GET request to "/products" with query parameters. It extracts the query parameters using `req.query` and handles retrieving products based on the provided category and price range.

By defining routes using different HTTP methods and URL patterns, you can handle various types of requests and extract data from the incoming requests using route parameters (`req.params`) and query parameters (`req.query`).

Middleware:

Middleware functions in Express are functions that have access to the request (`req`) and response (`res`) objects and can modify them or perform additional actions before passing control to the next middleware function in the chain. Middleware functions are a powerful mechanism for extending the functionality of your Express application.

Express provides several built-in middleware functions that can be easily incorporated into your application. Here are some commonly used built-in middleware functions:

1. **Body Parsing Middleware:** Express provides middleware functions for parsing different types of request bodies, such as JSON, URL-encoded, and multipart forms. These middleware functions populate the `req.body` property with the parsed data.

```
const express = require('express');
const app = express();
// Parse JSON bodies
app.use(express.json());
// Parse URL-encoded bodies
app.use(express.urlencoded({ extended: false }));
// Parse multipart form data
app.use(express.multipart());
```

2. **Logging Middleware:** Logging middleware functions help in logging incoming requests and related information. They can be used for debugging, monitoring, or auditing purposes.

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

3. **Error Handling Middleware:** Express provides middleware functions for handling errors that occur during the request-response cycle. These middleware functions are typically defined at the end of the middleware chain and are triggered when an error is thrown or when `next(err)` is called.

```
const errorHandler = (err, req, res, next) => {
 console.error(err);
 res.status(500).json({ error: 'Internal Server Error' });
};
app.use(errorHandler);
```

Apart from the built-in middleware functions, you can also create custom middleware to meet your application's specific requirements. Custom middleware functions can be used to add authentication, authorisation, request validation, logging, or any other custom logic.

```
const authenticate = (req, res, next) => {
 // Perform authentication logic here
 if (req.user) {
 next();
 } else {
 res.status(401).json({ error: 'Unauthorized' });
 }
};
app.use(authenticate);
```

To use middleware functions, you can call `app.use()` or `app.METHOD()` (e.g., `app.get()`, `app.post()`) with the middleware function as an argument. Middleware functions can be used globally for all routes or selectively for specific routes.

Remember to call `next()` within your middleware functions to pass control to the next middleware in the chain. If `next()` is not called, the request-response cycle will be halted, and the client will not receive a response.

Middleware functions in Express allow you to modularise and extend the functionality of your application, enabling you to handle common tasks, implement custom logic, and enhance the overall request-response flow.

Request and Response Handling:

In Express, you can handle incoming requests and send appropriate responses by accessing request data, such as headers and query parameters, parsing request data, and sending different types of responses like JSON, HTML, or files. Here's an overview of how you can achieve these tasks:

1. **Accessing Headers and Query Parameters:**
— Request Headers: You can access request headers using `req.headers`. For example, to access the `Content-Type` header: `req.headers['content-type']`.
— Query Parameters: Express provides access to query parameters through the `req.query` object. For example, for a URL like `/users?id=123`, you can access the `id` parameter using `req.query.id`.

2. **Parsing Request Data:**
— JSON Body: Express provides built-in middleware (`express.json()`) to parse JSON request bodies. Once parsed, the data is available in `req.body`. You need to use the middleware before your routes: `app.use(express.json())`.
— URL-Encoded Body: Express also provides middleware (`express.urlencoded()`) to parse URL-encoded form data. Similarly, the parsed data is available in `req.body`. Use it before your routes: `app.use(express.urlencoded({ extended: false }))`.

3. **Sending Responses:**
— JSON Response: To send a JSON response, you can use `res.json(data)`. For example, `res.json({ message: 'Success', data })`. Express automatically sets the appropriate `Content-Type` header.
— HTML Response: Use `res.send(htmlContent)` or `res.sendFile(filePath)` to send HTML responses or serve static HTML files.
— File Download: To initiate a file download, use `res.download(filePath)` and provide the file path.
— Redirect: Use `res.redirect(url)` to redirect the client to a different URL.
— Response Headers: You can set custom response headers using `res.set(header, value)` or `res.setHeader(header, value)`. For example, `res.set('Content-Disposition', 'attachment; filename="file.txt"')`.

Here's an example that demonstrates handling a POST request with JSON data and sending a JSON response:

```
const express = require('express');
const app = express();
app.use(express.json()); // Parse JSON bodies
app.post('/users', (req, res) => {
 const { name, email } = req.body;
// Perform logic with the received data
res.status(201).json({ message: 'User created', user: { name, email } });
});
app.listen(3000, () => {
 console.log('Server started on port 3000');
});
```

In this example:
- The `express.json()` middleware is used to parse JSON request bodies.
- The `app.post('/users', ...)` route handles the POST request to `/users` and extracts the `name` and `email` properties from `req.body`.

- After processing the data, a JSON response is sent with the status code `201` and a response body containing a success message and the user details.

By accessing request data, parsing request bodies, and sending appropriate responses, you can handle different types of requests and customise your application's behaviour based on the received data.

Error Handling:

Handling errors in Express applications involves catching and handling errors that occur during the request-response cycle, creating custom error handling middleware, and sending appropriate error responses to clients. Here's an overview of the techniques you can use:

1. **Catching Errors:** You can catch errors within route handlers using `try-catch` blocks or by using `async`/`await` with `try-catch` blocks. This allows you to handle errors within a specific route.

```
app.get('/users', async (req, res) => {
 try {
 // Perform an asynchronous operation
 const users = await fetchUsers();
res.json(users);
 } catch (error) {
 // Handle the error
 console.error(error);
 res.status(500).json({ error: 'Internal Server Error' });
 }
});
```

2. **Error Handling Middleware:** You can create custom error handling middleware to centralize error handling logic and keep your route handlers clean. This middleware should be defined after other routes and middleware, typically at the end of your middleware chain. It takes four parameters: `err`, `req`, `res`, and `next`. You can use this middleware to handle errors and send appropriate error responses.

```
app.use((err, req, res, next) => {
 // Handle the error
 console.error(err);
 res.status(500).json({ error: 'Internal Server Error' });
});
```

3. **Asynchronous Error Handling:** When working with asynchronous operations, such as database queries or API calls, you can use middleware like `async-middleware` or `express-async-handler` to handle errors automatically without needing to wrap each route handler in a `try-catch` block.

```
const asyncHandler = require('express-async-handler');
app.get('/users', asyncHandler(async (req, res) => {
 // Perform an asynchronous operation
 const users = await fetchUsers();
res.json(users);
}));
```

4. **Handling Synchronous Errors:** For synchronous operations, you can use the `next` function to pass the error to the error handling middleware. This is useful when dealing with synchronous functions or libraries that throw errors.

```
app.get('/users', (req, res, next) => {
 try {
 // Perform a synchronous operation
 const result = someSyncFunction();
if (result) {
 res.json(result);
 } else {
 throw new Error('Failed to get users');
 }
 } catch (error) {
 next(error); // Pass the error to the error handling middleware
 }
});
```

By using these techniques, you can effectively handle errors that occur within your Express application. Catching errors within route handlers, creating custom error handling middleware, and sending appropriate error responses allow you to gracefully handle errors and provide meaningful feedback to clients when something goes wrong.

Database Integration:

To connect and interact with databases in your Express application, you can use popular libraries like Mongoose for MongoDB. Mongoose provides a simple and elegant way to handle data persistence and perform CRUD (Create, Read, Update, Delete) operations. Here's an overview of using Mongoose in your Express application:

1. **Installing Mongoose**: Start by installing Mongoose as a dependency in your project. Run `npm install mongoose` to install it.

2. **Connecting to the Database**: In your Express application, establish a connection to your MongoDB database using Mongoose. You'll typically do this in your main app file or a separate database configuration file.

```
const mongoose = require('mongoose');
// Connect to the MongoDB database
mongoose.connect('mongodb://localhost/my-database', {
 useNewUrlParser: true,
 useUnifiedTopology: true,
})
 .then(() => {
 console.log('Connected to the database');
 })
 .catch((error) => {
 console.error('Error connecting to the database:', error);
 });
```

Replace `'mongodb://localhost/my-database'` with your actual MongoDB connection string.

3. **Creating Mongoose Schemas:** Define Mongoose schemas to represent the structure of your data models. Schemas define the fields and their types, along with any validation or configuration options.

```
const mongoose = require('mongoose');
const userSchema = new mongoose.Schema({
 name: String,
 email: {
 type: String,
 unique: true,
 required: true,
 },
 age: {
 type: Number,
 min: 18,
 },
});
const User = mongoose.model('User', userSchema);
```

In this example, we define a `User` schema with fields `name`, `email`, and `age`.

4. **Performing CRUD Operations:** Use Mongoose models to perform CRUD operations on your MongoDB database. Mongoose provides methods like `create()`, `find()`, `findOne()`, `updateOne()`, `deleteOne()`, etc., to interact with the database.

```
// Creating a new user
const user = new User({
 name: 'John Doe',
 email: 'john@example.com',
 age: 25,
});
user.save()
 .then(() => {
 console.log('User created');
 })
 .catch((error) => {
 console.error('Error creating user:', error);
 });
// Finding users
User.find()
 .then((users) => {
 console.log('Users:', users);
 })
 .catch((error) => {
 console.error('Error finding users:', error);
 });
// Updating a user
User.updateOne({ name: 'John Doe' }, { age: 26 })
 .then(() => {
 console.log('User updated');
 })
 .catch((error) => {
 console.error('Error updating user:', error);
 });
// Deleting a user
User.deleteOne({ name: 'John Doe' })
 .then(() => {
 console.log('User deleted');
 })
 .catch((error) => {
 console.error('Error deleting user:', error);
 });
```

These examples show how to create, find, update, and delete users using Mongoose methods.

By using Mongoose in your Express application, you can easily connect to your MongoDB database, define data models using schemas, and perform CRUD operations for data persistence.

Authentication and Authorisation:

Here are some code examples to illustrate the implementation of user authentication and authorisation in an Express application:

1. **Username/Password Authentication** with bcrypt for password hashing:

```
const bcrypt = require('bcrypt');
const User = require('../models/user');
// User login
app.post('/login', async (req, res) => {
 const { username, password } = req.body;
 const user = await User.findOne({ username });
if (!user) {
 return res.status(401).json({ message: 'Invalid credentials' });
 }
const isPasswordValid = await bcrypt.compare(password, user.password);
if (!isPasswordValid) {
 return res.status(401).json({ message: 'Invalid credentials' });
 }
// Generate and send authentication token
 const token = generateAuthToken(user);
 res.json({ token });
});
```

2. **Token-based Authentication** using JSON Web Tokens (JWT):

```
const jwt = require('jsonwebtoken');
const User = require('../models/user');
// User login
app.post('/login', async (req, res) => {
 const { username, password } = req.body;
 const user = await User.findOne({ username });
if (!user) {
 return res.status(401).json({ message: 'Invalid credentials' });
 }
const isPasswordValid = await user.comparePassword(password);
if (!isPasswordValid) {
 return res.status(401).json({ message: 'Invalid credentials' });
 }
// Generate and send authentication token
 const token = jwt.sign({ userId: user._id }, 'secret-key', { expiresIn: '1h' });
 res.json({ token });
});
```

3. **Middleware for Authentication and Authorisation:**

```
// Authentication middleware
function authenticate(req, res, next) {
 const token = req.headers.authorization?.split(' ')[1];
if (!token) {
 return res.status(401).json({ message: 'Authentication failed' });
 }
try {
 const decoded = jwt.verify(token, 'secret-key');
 req.userId = decoded.userId;
 next();
 } catch (error) {
 return res.status(401).json({ message: 'Invalid token' });
 }
}
// Authorisation middleware
function authorise(role) {
 return (req, res, next) => {
 const userRole = getUserRole(req.userId);
if (userRole !== role) {
 return res.status(403).json({ message: 'Access denied' });
 }
next();
 };
}
// Protected route with authentication and authorisation
app.get('/admin/dashboard', authenticate, authorise('admin'), (req, res) => {
 res.json({ message: 'Welcome to the admin dashboard' });
});
```

These examples demonstrate the basic implementation of user authentication and authorisation in an Express application using different techniques. However, it's important to note that the actual implementation may vary depending on your specific application requirements and architecture.

What is the… Next function argument?

In Express, `next` is a function that is passed as an argument to middleware functions. It is used to pass control from one middleware function to the next middleware function in the chain. By invoking `next()`, the current middleware function signals that it has completed its processing and wants to pass the control to the next middleware function or route handler.

The `next` function can be called in different ways, each with a different effect on the request-response

request.

2. `next('route')`: Calling `next('route')` skips the remaining middleware functions in the current chain and moves to the next matching route handler. It effectively jumps to the next route handler, bypassing any remaining middleware functions for the current route.

3. `next(error)`: Calling `next` with an error object as an argument triggers the error handling middleware. Express recognises this as an error and skips all remaining middleware functions, passing control to the

error handling middleware registered using `app.use(errorHandler)` or `app.use(function(err, req, res, next) { ... })`.

The `next` function is commonly used to create middleware chains in Express applications. Each middleware function has the option to either pass control to the next middleware by invoking `next()`, modify the request or response objects, or terminate the request-response cycle by sending a response or invoking the error handling middleware.

It's important to note that if the `next` function is not called within a middleware function, the request will hang indefinitely, and the client will not receive a response. Therefore, it's crucial to ensure that `next` is called appropriately in each middleware function to maintain the flow of the request-response cycle.

What kind of.... Express functions do we have available to us?

Express comes with several built-in functions and methods that facilitate the development of web applications. Here are some commonly used functions in Express:

1. `express()`: This is the top-level function used to create an Express application. It returns an instance of the Express application that can be used to configure routes, middleware, and other settings.

2. `app.use()`: This method is used to mount middleware functions in the application's request-response cycle. It can be used to apply middleware globally to all routes or selectively to specific routes.

3. `app.METHOD()`: These methods (`app.get()`, `app.post()`, `app.put()`, `app.delete()`, etc.) are used to define route handlers for specific HTTP methods. They specify the callback function to be executed when a request matching the specified method and route is received.

4. `app.set()`: This method is used to set application-level settings or configuration variables. It allows you to define various settings, such as the view engine, port number, or any custom settings you need.

5. `app.get()` and `app.set()`: These methods are used to retrieve and set application-level settings, respectively. They can be used to access and modify the values set using `app.set()`.

6. `app.listen()`: This method is used to start the Express application and listen for incoming requests on a specified port. It takes a port number and an optional callback function to be executed when the server starts.

7. `app.route()`: This method provides a convenient way to define multiple routes for a single path prefix. It allows you to chain multiple HTTP methods to a single route handler.

8. `app.param()`: This method is used to define route parameters that are common to multiple routes. It enables you to define a middleware function that will be executed whenever a specific parameter is present in the route path.

9. `res.send()`: This method is used to send a response back to the client. It can send various types of responses, including plain text, HTML, JSON, or even files.

10. `res.render()`: This method is used to render and send a response using a specified template engine. It is commonly used to render dynamic HTML views by passing data to the view template.

11. \`req.params\`, \`req.query\`, \`req.body\`: These properties of the \`req\` object provide access to route parameters, query parameters, and the request body, respectively. They allow you to extract data from incoming requests.

These are just a few of the functions provided by Express. Express is highly extensible, and you can also use additional middleware functions and third-party libraries to enhance the functionality of your application.

Expressjs

M

Follow

Written by Marazzo

1 Follower · 3 Following

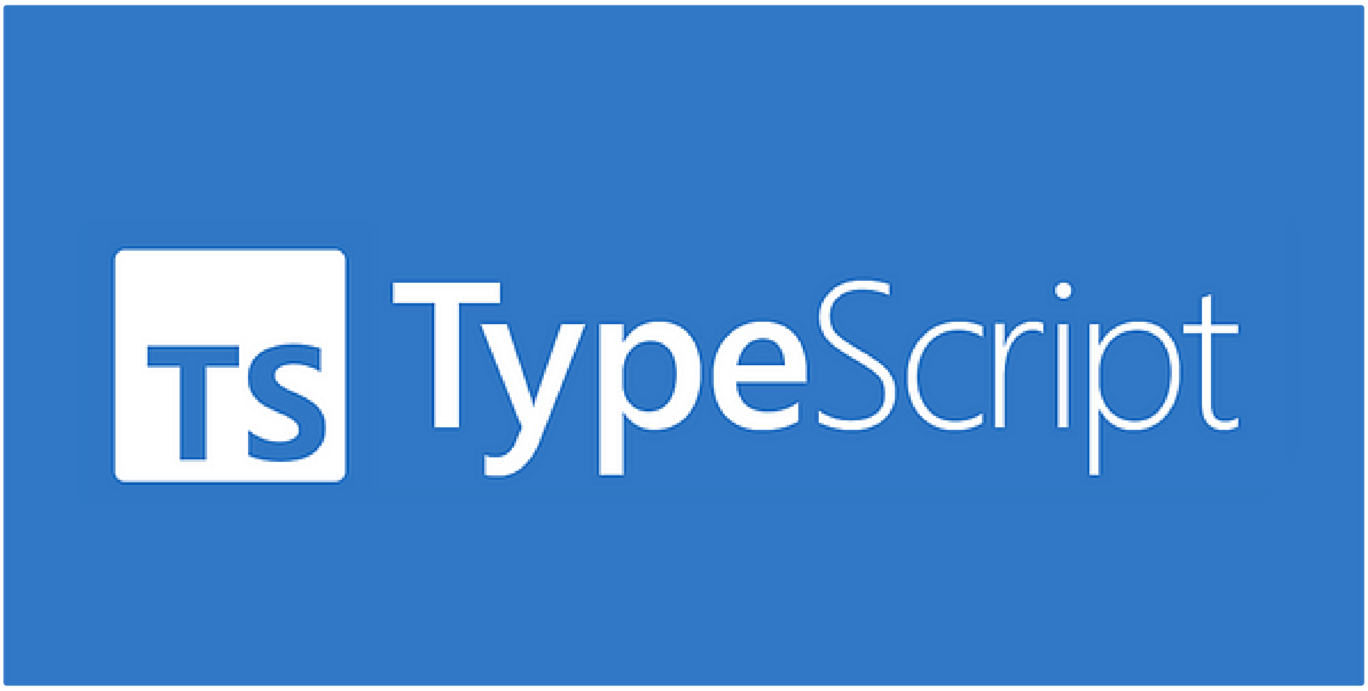
Apr 27, 2023 26

Marazzo

Understanding Prototypal Inheritance in JavaScript

JavaScript is an object-oriented language that supports prototypal inheritance. Prototypal inheritance is a way for objects to inherit…

Apr 27, 2023

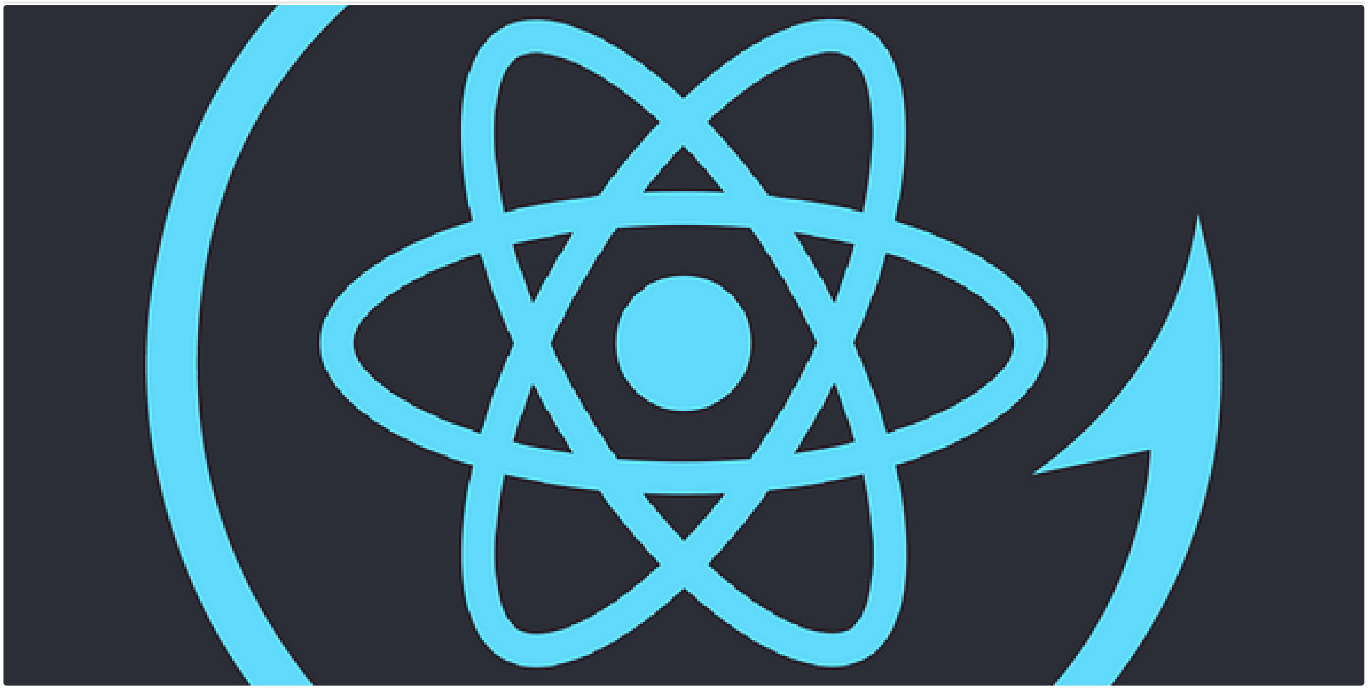


Marazzo

Typescript — Beginner

Part 1: TypeScript Basics

Apr 26, 2023



Marazzo

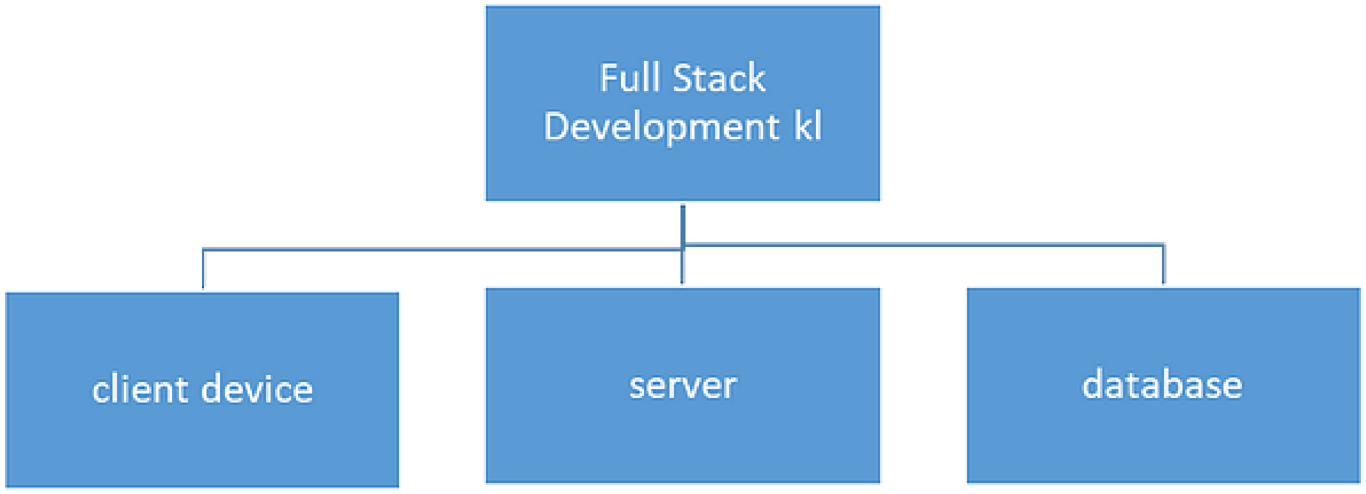
React Hooks — Core Hooks

Introduction

Apr 26, 2023 4

See all from Marazzo

Recommended from Medium



Vidhyapramila

Full Stack (MERN stack) Development

JavaScript Basics :

2d ago

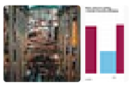
habtesoft

100 NodeJS Questions for Technical Interviews

Whether you're gearing up for a tech interview or brushing up on your Node.js knowledge, this comprehensive list of 100 Node.js questions...

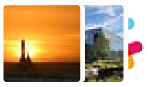
Aug 23 4

Lists



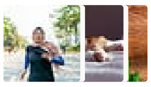
Staff picks

776 stories · 1467 saves



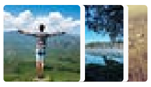
Stories to Help You Level-Up at Work

19 stories · 879 saves



Self-Improvement 101

20 stories · 3089 saves



Productivity 101

20 stories · 2596 saves

Samuelnoye

A Beginner's Guide to Using Node.js with MongoDB

This guide will walk you through the steps of setting up a Node.js application with MongoDB, including examples to get you started.

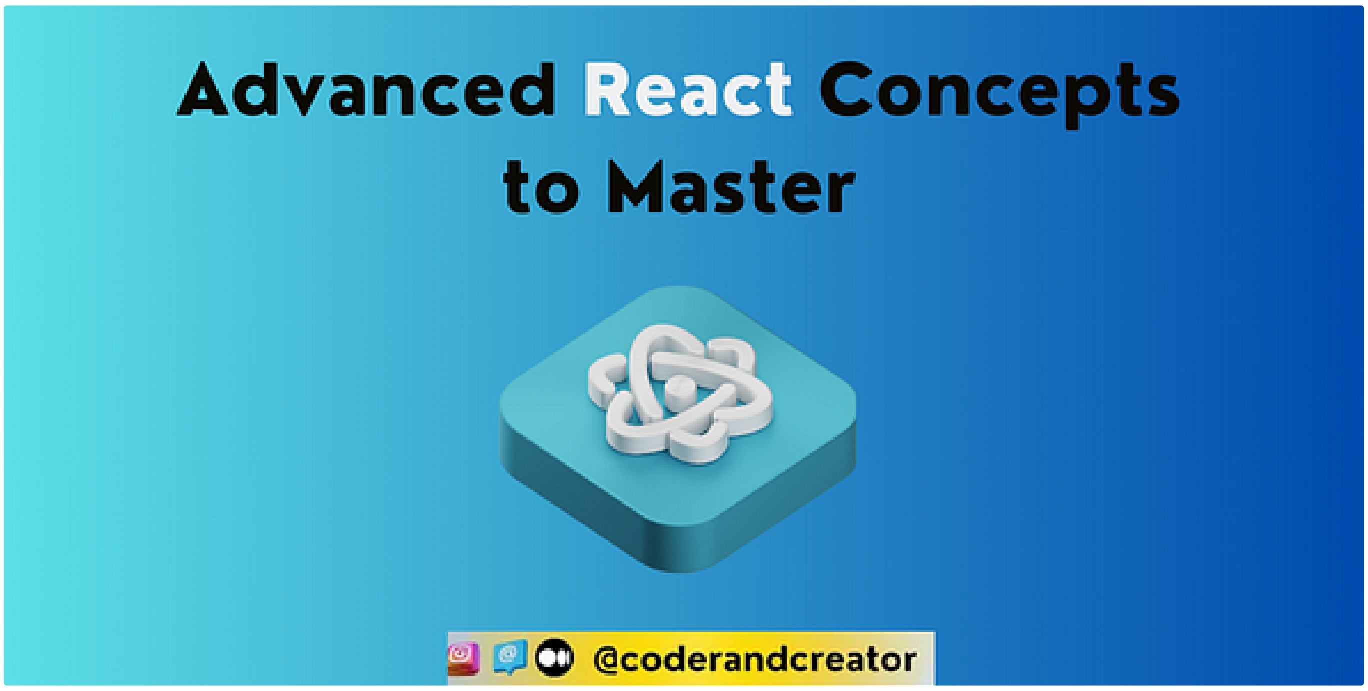
Jun 7

kelvinBz

MERN — ExpressJS Fundamentals

ExpressJS Fundamentals

4d ago

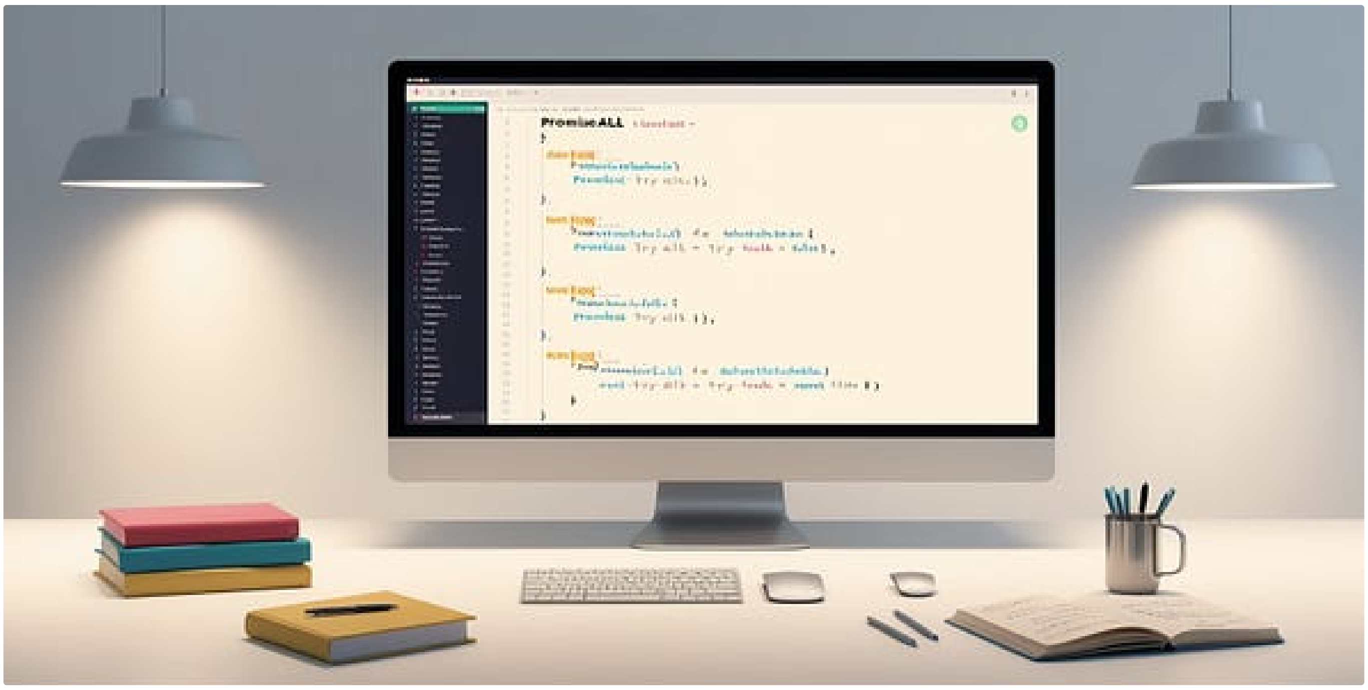


In Stackademic by coderandcreator

Advanced React Concepts to Master Before Your Dream Interview

Are you a seasoned React developer looking to take your skills to the next level? Or perhaps you're preparing for a challenging interview...

Oct 28 110 2



Mayur Koshti

Using Promise all, try-catch, and await in React

React, a popular JavaScript library for building user interfaces often requires handling asynchronous operations.

Sep 30

See more recommendations