Building Web Applications with Express.js: A Comprehensive Guide

Learn how to build scalable and maintainable web applications with Express.js, the popular Node.js web framework. Our comprehensive guide covers everything from setting up your development environment to creating and deploying your first app.

Shahzaib Khan · Follow
15 min read · Mar 1, 2023

Listen Share

Express.js is a popular framework for building web applications on top of Node.js. It provides a robust set of features and tools that make it easy to develop scalable and maintainable web applications. In this post, we will explore the various components of Express.js and learn how to build a basic web application using it. To summaries, you will learn:

What is Express.js?

Express.js is a popular web application framework for Node.js. It provides a set of tools and features for building web applications and APIs quickly and easily. It simplifies the process of building web applications by providing a robust set of features for handling HTTP requests and responses, routing, middleware, templates and views, serving static files, and more. Express.js is widely used in the Node.js community and has a large ecosystem of plugins and extensions available for added functionality.

Setting up a basic Express.js application

To set up a basic Express.js application, follow these steps:

1. Create a new folder for your project and navigate to it in your terminal.
2. Initialize a new Node.js project using `npm init` command and follow the prompts to create a `package.json` file for your project.
3. Install Express.js as a dependency using the `npm install express` command.
4. Create a new JavaScript file in the project directory and name it `app.js` or any name you prefer.
5. Open `app.js` file in your code editor and add the following code:

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

app.get('/', (req, res) => {
```

```
  res.send('Hello, World!');
});

app.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

In the above code, we have required the `express` module and created a new Express application by calling the `express()` function, which returns an instance of the application.

We have then defined a route for the root URL `/` using the `app.get()` method. When the root URL is requested, the server will send the response `Hello, World!`.

Finally, we have started the server by calling the `app.listen()` method and passing in the port number that the server should listen on. In this case, the server will listen on port 3000.

1. Save the changes and run the `node app.js` command in your terminal to start the server.
2. Open your web browser and navigate to `http://localhost:3000` to see the "Hello, World!" message displayed in your browser.

Congratulations, you have now set up a basic Express.js application!

Routing in Express.Js

Routing in Express.js refers to the process of defining application end points (URIs) and how requests and responses should be handled at each endpoint. Routing in Express.js can be done using the `express.Router()` method, which creates a new router object.

Here is an example of how to define a basic route in an Express.js application:

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

// Define a route
app.get('/hello', (req, res) => {
  res.send('Hello, world!');
});
```

```
// Start the server
app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

In the above example, we have defined a route that listens for GET requests to the `/hello` endpoint. When a request is made to this endpoint, the server will respond with the message "Hello, world!".

Routing in Express.js can be more complex than this, however. You can define routes that accept parameters, use middleware, and more. Here is an example of a more complex route definition:

```
app.get('/users/:id', (req, res, next) => {
  // Retrieve the user from the database
  const user = db.getUserById(req.params.id);

  // If the user is not found, pass control to the error handler
  if (!user) {
    return next(new Error('User not found'));
  }

  // Render the user's profile page
  res.render('user', { user });
});
```

In this example, we are defining a route that accepts a parameter `id` in the URL. We then retrieve the user with that ID from the database, and render a template with the user's information. If the user is not found, we pass control to the error handler middleware by calling `next()` with an error object.

Routing is a fundamental concept in building web applications with Express.js, and is used extensively throughout the framework.

Using Middleware in Express.Js

Middleware in Express is a function that acts as a bridge between an incoming request and the route handler. It allows you to execute some logic before the request is sent to the route handler. For example, you can use middleware to authenticate users, parse data, or add headers to the response.

Here's an example of how to use middleware in Express:

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

// Middleware function
const logRequest = (req, res, next) => {
  console.log(`Received a ${req.method} request from ${req.ip}`);
  next();
};

// Use the middleware
app.use(logRequest);

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

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

In this example, we define a middleware function `logRequest` that logs a message to the console every time a request is made. We use the `app.use()` method to attach the middleware to the Express app. When a request is received, the middleware function is executed first, then the route handler is executed.

There are several other ways to use middleware in Express, such as attaching it to a specific route, or a group of routes. You can also apply middleware globally to all requests by using `app.use()` without a specific path.

Is it essential to use middleware? and what are more use cases of them?

Using middleware is not essential, but it's a very useful feature that can greatly enhance the functionality of your Express application. Here are some common use cases for middleware:

1. **Authentication**: You can use middleware to authenticate users before allowing them to access certain routes.
2. **Data parsing:** You can use middleware to parse incoming data, such as JSON or form data, so that it can be easily accessed in your route handlers.

3. **Error handling**: You can use middleware to handle errors that occur during the processing of requests.
4. **Logging**: You can use middleware to log information about requests, such as the request method, IP address, and request/response time.
5. **CORS**: You can use middleware to handle Cross-Origin Resource Sharing (CORS) and set appropriate headers to control who can access your API.
6. **Static file serving**: You can use middleware to serve static files, such as images, CSS, and JavaScript files, from your Express application.

In general, middleware can be used for any logic that needs to be executed before the route handler is called. It's a powerful feature that makes it easy to modularize and reuse your code, making it easier to maintain and scale your application.

Templates and Views

In Express.js, templates and views are used to render dynamic content and display it to the user in a structured way. Templates are HTML files that contain placeholders for dynamic content, while views are templates that are dynamically rendered with data to generate HTML. In this way, we can create dynamic web pages with reusable templates.

To use templates and views in Express.js, we need a view engine. A view engine is a module that can parse templates and generate HTML output. There are many view engines available for Express.js, such as EJS, Pug, Handlebars, and Mustache.

Here's an example of how to use the EJS view engine in an Express.js application:

Considering you have already setup the basic express.js application, we now install install the EJS module using npm:

```
npm install ejs
```

Next, create an `index.ejs` file in a `views` directory in your project:

```
<!-- views/index.ejs -->

<!DOCTYPE html>
<html>
  <head>
    <title>Home Page</title>
  </head>
  <body>
    <h1>Welcome to the Home Page</h1>
    <p>This is some content for the home page.</p>
  </body>
</html>
```

Then, create an `about.ejs` file in the same `views` directory:

```
<!-- views/about.ejs -->

<!DOCTYPE html>
<html>
  <head>
    <title>About Page</title>
  </head>
  <body>
    <h1>About Us</h1>
    <p>This is some content for the about page.</p>
  </body>
</html>
```

In your `app.js` file, set the view engine to EJS and define routes for the home page and the about page:

```
// app.js

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

// Set view engine to EJS
app.set('view engine', 'ejs');
app.set('views', path.join(__dirname, 'views'));

// Define routes
```

```
app.get('/', function(req, res) {
  res.render('index');
});

app.get('/about', function(req, res) {
  res.render('about');
});

// Start server
app.listen(3000, function() {
  console.log('Server started on port 3000');
});
```

Now, start the server and navigate to `http://localhost:3000` to see the home page, or `http://localhost:3000/about` to see the about page. Express.js will automatically render the `index.ejs` or `about.ejs` file, depending on the requested route.

Using the above example and making dynamic templates that can take the data and show case:

Here's an example that uses EJS as the view engine and passes data to the home page and about page:

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

// Set EJS as the view engine
app.set('view engine', 'ejs');
app.set('views', path.join(__dirname, 'views'));

// Home page route
app.get('/', (req, res) => {
  const data = {
    title: 'Home Page',
    message: 'Welcome to the home page!'
  }
  res.render('index', { data });
});

// About page route
app.get('/about', (req, res) => {
  const data = {
    title: 'About Page',
    message: 'Learn more about us!'
```

```
  }
  res.render('about', { data });
});

// Listen on the port
app.listen(port, () => console.log(`App listening on port ${port}`));
```


view file (e.g. `home.ejs` or `about.ejs`).

Inside the view files, we can access the data using the `data` variable. Here's an example of what the `home.ejs` file might look like:

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= data.title %></title>
  </head>
  <body>
    <h1><%= data.message %></h1>
  </body>
</html>
```

Similarly, the `about.ejs` file might look like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= data.title %></title>
  </head>
  <body>
    <h1><%= data.message %></h1>
  </body>
</html>
```

When the user visits the home page, they will see a message that says “Welcome to the home page!” with the title “Home Page” in the browser tab. When they visit the

about page, they will see a message that says “Learn more about us!” with the title “About Page” in the browser tab.

Serving static files in Express.Js

In Express.js, serving static files such as images, CSS, and JavaScript files is a common task. Express.js makes it easy to serve static files using the `express.static()` middleware function.

The `express.static()` function takes one argument, which is the directory from where the static files should be served. Once the middleware function is set up, any file in the specified directory can be accessed from the browser with the URL path.

Here’s an example of how to set up the `express.static()` middleware function in an Express.js app:

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

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

// start the server
app.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

In the example above, the `express.static()` function is used to serve files from the `public` directory. This means that any file in the `public` directory can be accessed from the browser with the URL path, for example: `http://localhost:3000/images/logo.png`.

It’s also possible to serve static files from multiple directories by calling the `express.static()` middleware function multiple times, passing a different directory each time.

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

// serve static files from the public directory
```

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

// serve static files from the assets directory
app.use(express.static('assets'));

// start the server
app.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

In the example above, files from both the `public` and `assets` directories can be accessed from the browser.

Serving static files with Express.js is a simple and efficient way to serve static assets for your web application.

Handling forms and input data in Express.Js

In a web application, forms are commonly used to collect input data from users. Express.js provides built-in functionality to handle forms and input data through middleware. The middleware used for handling forms and input data in Express.js is `body-parser`.

`body-parser` middleware parses the data submitted through a form and makes it available in the `req.body` object.

To use `body-parser`, it must be installed and required in the application:

```
npm install body-parser
```

Now let's use it:

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
```

```
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
```

`body-parser` can handle different types of data. The `urlencoded` function of `body-parser` is used to parse data submitted in a form using the `POST` method. The `json` function is used to parse JSON data.

After `body-parser` is set up, it can be used in a route handler to handle form submission:

```
app.post('/submit-form', (req, res) => {
  const formData = req.body;
  console.log(formData);
  res.send('Form submitted successfully');
});
```

In this example, a route handler is set up to handle the `POST` request made when a form is submitted. The data submitted in the form is available in the `req.body` object. In this case, the data is logged to the console and a response is sent to the client.

Other types of input data, such as files, can also be handled using middleware such as `multer`.

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

const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, 'uploads/')
  },
  filename: function (req, file, cb) {
    cb(null, file.fieldname + '-' + Date.now())
  }
})

const upload = multer({ storage: storage })

app.post('/upload-file', upload.single('file'), (req, res) => {
  console.log(req.file);
```

```
  res.send('File uploaded successfully');
});
```

In this example, `multer` middleware is used to handle file uploads. The `upload` object is created with a configuration object that specifies where the files should be stored and how the files should be named. The `upload.single` function is used to specify that only one file should be uploaded. In this case, the file is expected to be submitted with the name `file`. The uploaded file is available in the `req.file` object. In this case, the file information is logged to the console and a response is sent to the client.

Cookies and sessions in Express.js

Cookies and sessions are used for maintaining state between HTTP requests. They are used to store data on the client's side and server-side, respectively. **Cookies are used to store data in the browser**, whereas **sessions are used to store data on the server-side.**

In Express.js, cookies and sessions are implemented using middleware.

To use cookies in Express.js, you need to install the cookie-parser middleware using NPM.

```
npm install cookie-parser
```

After installing the cookie-parser middleware, you can use it in your Express.js application by requiring it and using it as middleware.

```
const express = require('express');
const cookieParser = require('cookie-parser');

const app = express();

app.use(cookieParser());
```

Now, you can set and read cookies in your application. Here is an example of setting a cookie:

```
app.get('/set-cookie', (req, res) => {
  res.cookie('username', 'john');
  res.send('Cookie has been set');
});
```

Here, we are setting a cookie with the name `username` and the value `john`. The `res.cookie()` method is used to set cookies in the response.

To read cookies, you can use the `req.cookies` object. Here is an example:

```
app.get('/get-cookie', (req, res) => {
  const username = req.cookies.username;
  res.send(`The value of username cookie is ${username}`);
});
```

Here, we are reading the value of the `username` cookie and sending it as a response.

Sessions, on the other hand, require a bit more setup. To use sessions in Express.js, you need to install the express-session middleware using NPM.

```
npm install express-session
```

After installing the express-session middleware, you can use it in your Express.js application by requiring it and using it as middleware.

```
const express = require('express');
const session = require('express-session');

const app = express();

app.use(session({
```

```
  secret: 'your secret key',
  resave: false,
  saveUninitialized: true,
  cookie: { secure: false }
}));
```

Here, we are configuring the `express-session` middleware to use a secret key for encrypting session data. The `resave` option is set to `false` to prevent resaving the session data on every request, and the `saveUninitialized` option is set to `true` to save the session data even if it is empty. The `cookie` option is set to `{ secure: false }` to allow the session cookie to be sent over HTTP.

Now, you can set and read session data in your application. Here is an example of setting session data:

```
app.get('/set-session', (req, res) => {
  req.session.username = 'john';
  res.send('Session data has been set');
});
```

Here, we are setting a session variable with the name `username` and the value `john`. The `req.session` object is used to store session data.

To read session data, you can use the `req.session` object. Here is an example:

```
app.get('/get-session', (req, res) => {
  const username = req.session.username;
  res.send(`The value of username session variable is ${username}`);
});
```

Here, we are reading the value of the `username` session variable and sending it as a response.

Here's a simple example of a "Hello World" application that uses sessions and cookies in Express.js:

```
const express = require('express');
const cookieParser = require('cookie-parser');
const session = require('express-session');

const app = express();

app.use(cookieParser());
app.use(session({
    secret: 'mysecretkey',
    resave: true,
    saveUninitialized: true
}));

app.get('/', (req, res) => {
    let count = req.session.count || 0;
    res.cookie('name', 'John');
    req.session.count = ++count;
    res.send(`Hello ${req.cookies.name}, you have visited this page ${count} ti
});

app.listen(3000, () => {
    console.log('Server started on port 3000');
});
```

In this example, we first import the required modules: `express`, `cookie-parser`, and `express-session`. We then initialize an instance of the `express` application and use the `cookie-parser` and `express-session` middleware to handle cookies and sessions.

The `cookie-parser` middleware parses the cookies attached to the incoming HTTP request and makes them available on the `req.cookies` object.

The `express-session` middleware provides a way to manage user sessions in the application. In this example, we configure it with a secret key that is used to sign the session ID cookie, and set `resave` and `saveUninitialized` options to true.

We then define a single route for the root path `/`. Inside the route handler function, we first check if the session already exists by accessing the `count` property on the `req.session` object. If it doesn't exist, we initialize it with a value of 0. We then set a cookie named `name` with the value `John`. Finally, we increment the `count` property and send a response to the client with a personalized greeting that includes the cookie value and the number of times the user has visited the page.

When we run this application and open it in the browser, we should see a message that says “Hello John, you have visited this page 1 times!”. On subsequent visits, the count should increment and the message should reflect the new count.

This is a very basic example, but it demonstrates the use of sessions and cookies in an Express.js application. In a real-world application, you would likely use more sophisticated session management techniques and include error handling and security measures to protect against attacks such as cross-site scripting (XSS) and cross-site request forgery (CSRF).

User authentication in Express.js

User authentication is a crucial aspect of any web application that requires secure access control. It involves validating user identity and authorizing access to resources or functionalities based on user credentials.

Express.js provides various authentication mechanisms that can be integrated into an application, including:

1. Passport.js — A popular authentication middleware that supports various authentication strategies, including local authentication, OAuth, OpenID, and more.
2. JSON Web Tokens (JWT) — A standard for securely transmitting information between parties as a JSON object. It can be used to authenticate and authorize users by generating and validating tokens containing user information.
3. Session-based authentication — A traditional approach that involves creating and maintaining user sessions on the server-side.

For this post, we will be using session-based authentication. Make sure you have installed express.js and also express-session. Let’s start with the example:

```
const express = require('express');
const session = require('express-session');

const app = express();

// Use sessions middleware
app.use(session({
  secret: 'mySecret', // session secret
  resave: false,
```

```
  saveUninitialized: true,
  cookie: { secure: false }
}));

// Login route
app.post('/login', (req, res) => {
  const { username, password } = req.body;

  // Validate user credentials
  if (username === 'admin' && password === 'password') {
    // Set user session
    req.session.user = { username };
    res.send('Logged in successfully');
  } else {
    res.status(401).send('Invalid username or password');
  }
});

// Logout route
app.post('/logout', (req, res) => {
  // Destroy user session
  req.session.destroy((err) => {
    if (err) {
      console.error(err);
    } else {
      res.send('Logged out successfully');
    }
  });
});

// Protected route
app.get('/protected', (req, res) => {
  // Check if user is authenticated
  if (req.session.user) {
    res.send('Welcome to the protected area');
  } else {
    res.status(401).send('Unauthorized access');
  }
});

// Start the server
app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

In this example, we have used the `express-session` middleware to create and manage user sessions. The middleware adds a `session` object to the request object (`req`), which can be used to store and retrieve session data.

We have defined three routes:

1. `/login` - Authenticates the user by checking the credentials and creating a new session for the user.
2. `/logout` - Destroys the user session.
3. `/protected` - A protected route that can only be accessed by authenticated users.

When the user logs in successfully, we set a `user` object in the session that contains the user's username. When the user logs out, we destroy the session, which removes all session data.

In the protected route, we check if the user is authenticated by checking if the `user` object exists in the session. If the user is not authenticated, we return a 401 Unauthorized response.

Overall, user authentication is a critical aspect of web development, and using the right authentication mechanism can help ensure the security of your application.

Additional Links:

Get Started with Node.js: An Introduction to Node.js and its Basics

This post provides a comprehensive introduction to Node.js and covers the basics of how it works, how to install it...

medium.com

Mastering Node.js Basics: A Comprehensive Guide with Examples

Learn Node.js basics and improve your JavaScript skills with our comprehensive guide. Discover how to use Node.js...

medium.com

Thanks for reading! If you enjoy reading this post, got help, knowledge, inspiration, and motivation through it, and you want to support me — you can "buy me a coffee." Your support really makes a difference ❤️

Receive an **email** whenever I publish an article and consider being a **member** if you liked the story.

If you enjoyed this post…it would mean a lot to me if you could ***click on the "claps" icon…*** *up to 50 claps allowed — Thank You!*

Nodejs Expressjs Learning Nodejs

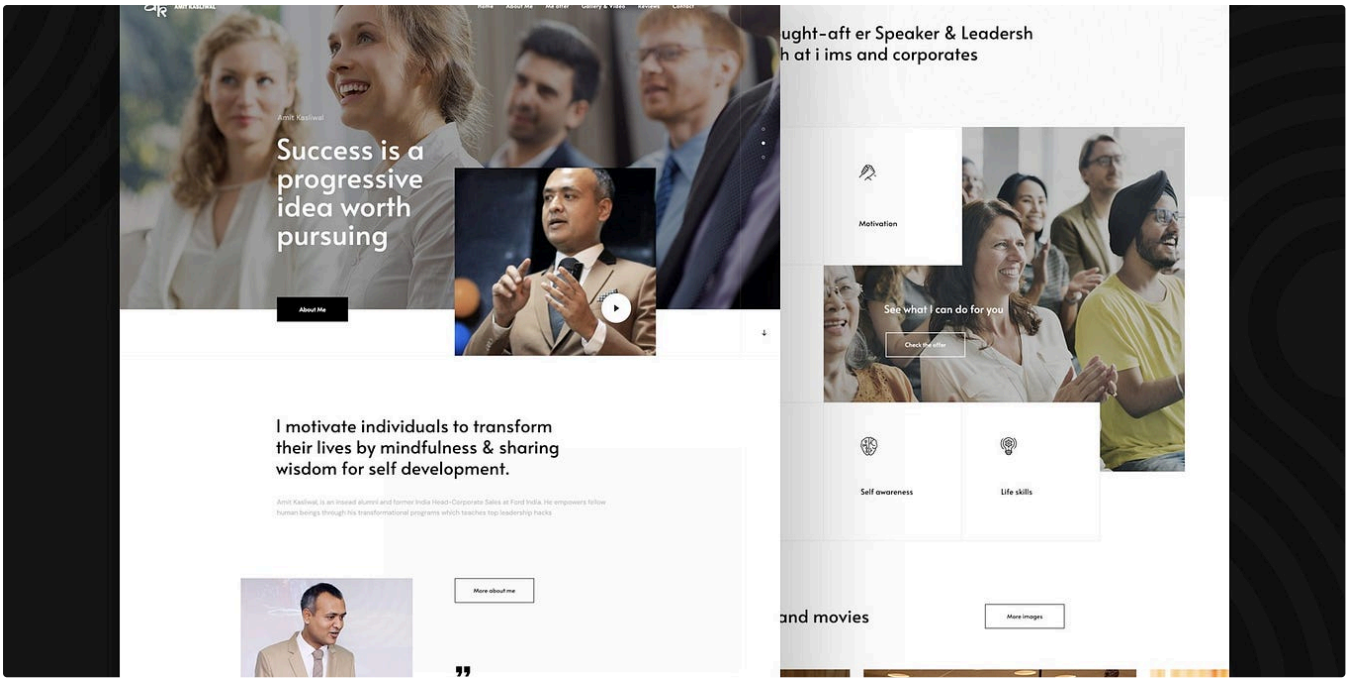
Follow

Written by Shahzaib Khan

321 Followers

Developer / Data Scientist / Computer Science Enthusiast. Founder @ Interns.pk You can connect with me @ https://linkedin.com/in/shahzaibkhan/

More from Shahzaib Khan



Shahzaib Khan

The 25 Best Personal Portfolio Website Design Inspiration & Examples

Personal websites play an important role in giving an individual the opportunity to tell their story.

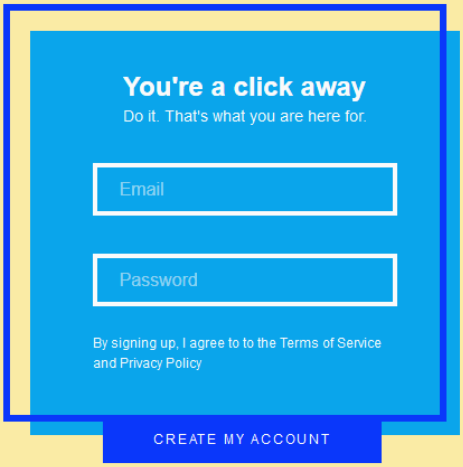
Jul 8, 2020 338 2

Shahzaib Khan

How to Build a Basic Node.js CRUD App with Mongoose and MongoDB

In this post, we will guide you through the process of building a basic Node.js application with Mongoose ORM and MongoDB database. We'll...

Mar 2, 2023 152 2



Shahzaib Khan

30+ Codepen's for your Login form layouts

Whether you are new to the world of web development or an accomplished developer, you might have come across CodePen.

Jul 9, 2020 14

Shahzaib Khan

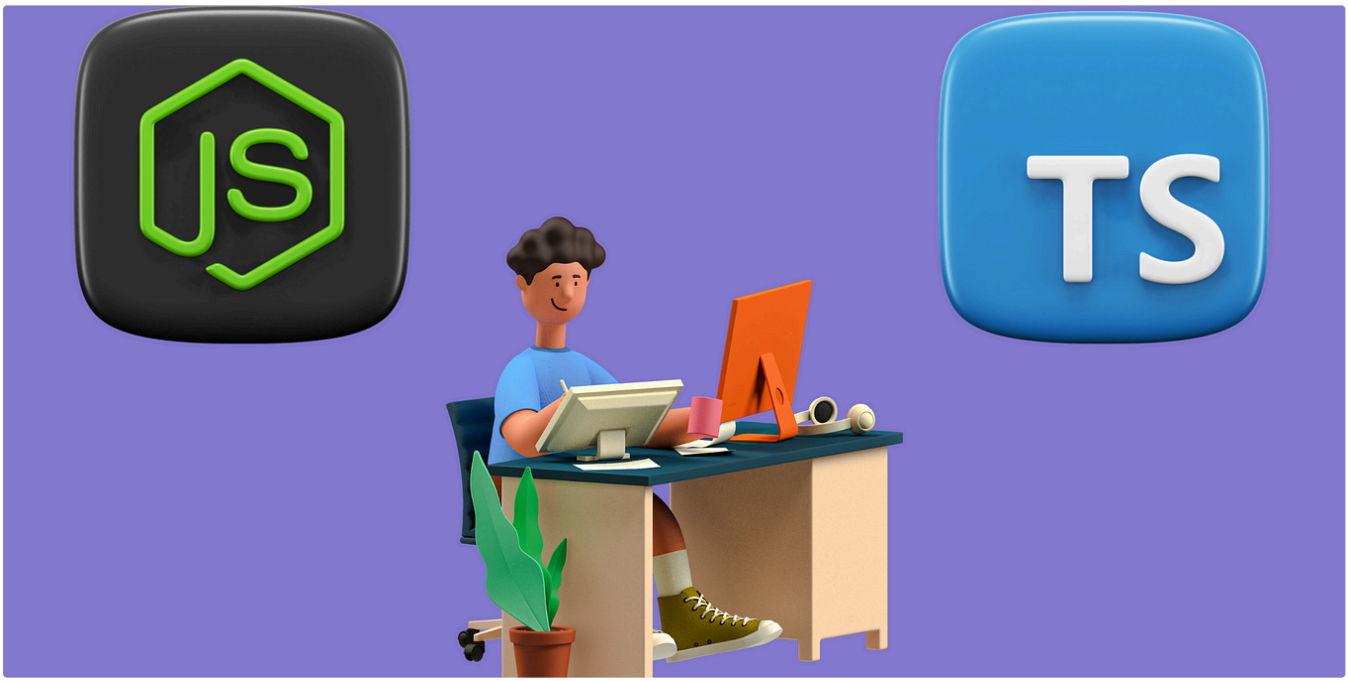
Building a RESTful API with Express.js: A Beginner's Guide

In this tutorial, we will walk through the process of building a RESTful API using Express.js, a popular Node.js web framework. We will...

Mar 3, 2023 15 1

See all from Shahzaib Khan

Recommended from Medium

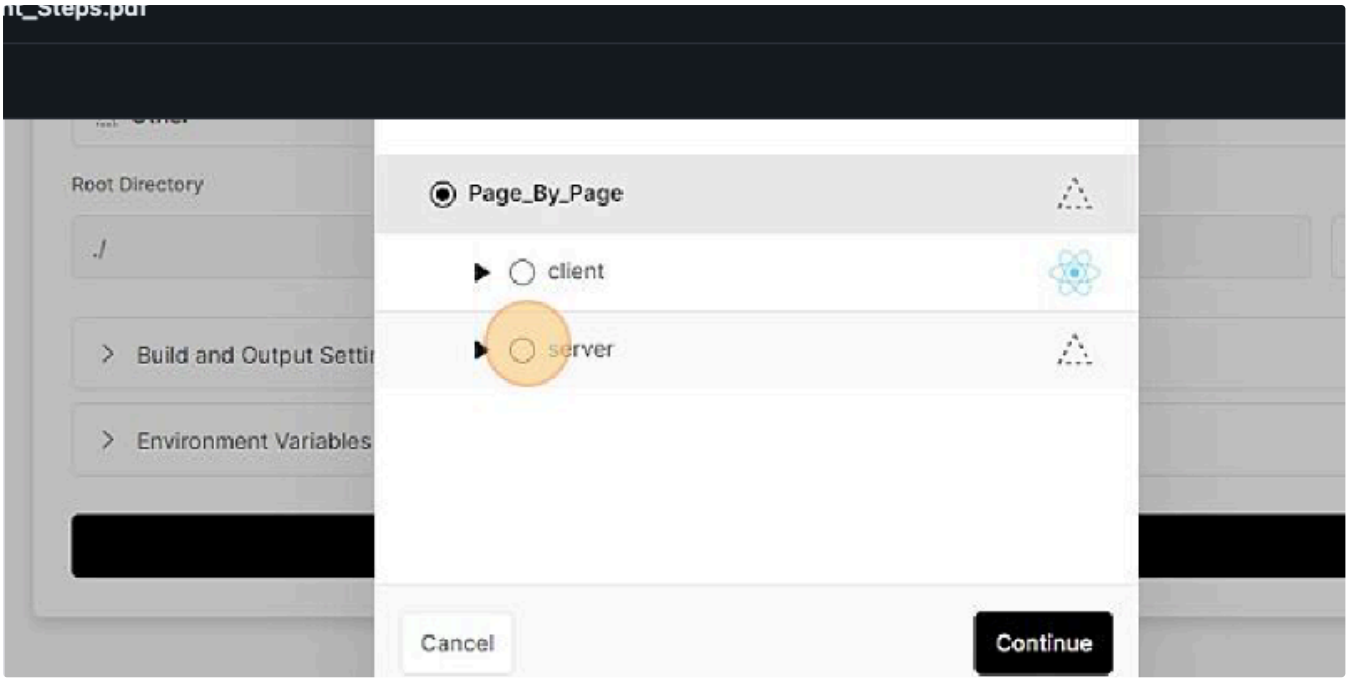


Pabath Induwara

A Step-by-Step Guide to Setting Up a Node.js Project with TypeScript

Integrating TypeScript with Node.js enhances the development process by improving code readability and maintainability. While it doesn't...

Feb 10 147



Dezarea Bryan

Deploying an Express Server on Vercel: A Step-by-Step Guide

Introduction

Apr 10 3

Lists

Stories to Help You Grow as a Software Developer
19 stories · 1232 saves

data science and AI
40 stories · 203 saves

Amazon.com Seattle, WA
Software Development Engineer Mar. 2020 – May 2021

- Developed Amazon checkout and payment services to handle traffic of 10 Million daily global transactions
- Integrated Iframes for credit cards and bank accounts to secure 80% of all consumer traffic and prevent CSRF, cross-site scripting, and cookie-jacking
- Led Your Transactions implementation for JavaScript front-end framework to showcase consumer transactions and reduce call center costs by $25 Million
- Recovered Saudi Arabia checkout failure impacting 4000+ customers due to incorrect GET form redirection

Projects

NinjaPrep.io (React)
- Platform to offer coding problem practice with built in code editor and written + video solutions in React
- Utilized Nginx to reverse proxy IP address on Digital Ocean hosts
- Developed using Styled-Components for 95% CSS styling to ensure proper CSS scoping
- Implemented Docker with Seccomp to safely run user submitted code with < 2.2s runtime

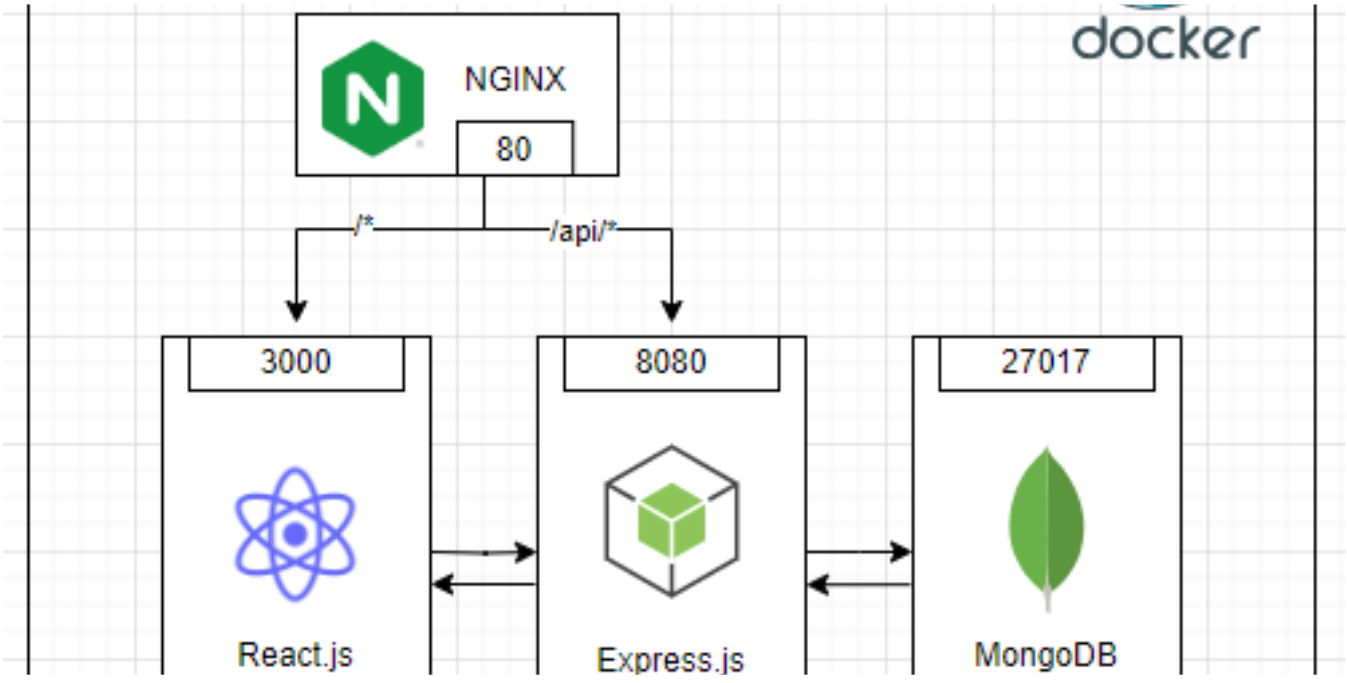
HeatMap (JavaScript)
- Visualized Google Takeout location data of location history using Google Maps API and Google Maps heatmap code with React
- Included local file system storage to reliably handle 5mb of location history data
- Implemented Express to include routing between pages and jQuery to parse Google Map and implement heatmap overlay

Alexander Nguyen in Level Up Coding

The resume that got a software engineer a $300,000 job at Google.

1-page. Well-formatted.

Jun 1 15.1K 231



IAMWYNN

Building a Production-Ready MERN Application

In this comprehensive tutorial, we'll walk you through the process of building a production-ready MERN (MongoDB, Express, React, Node.js)...

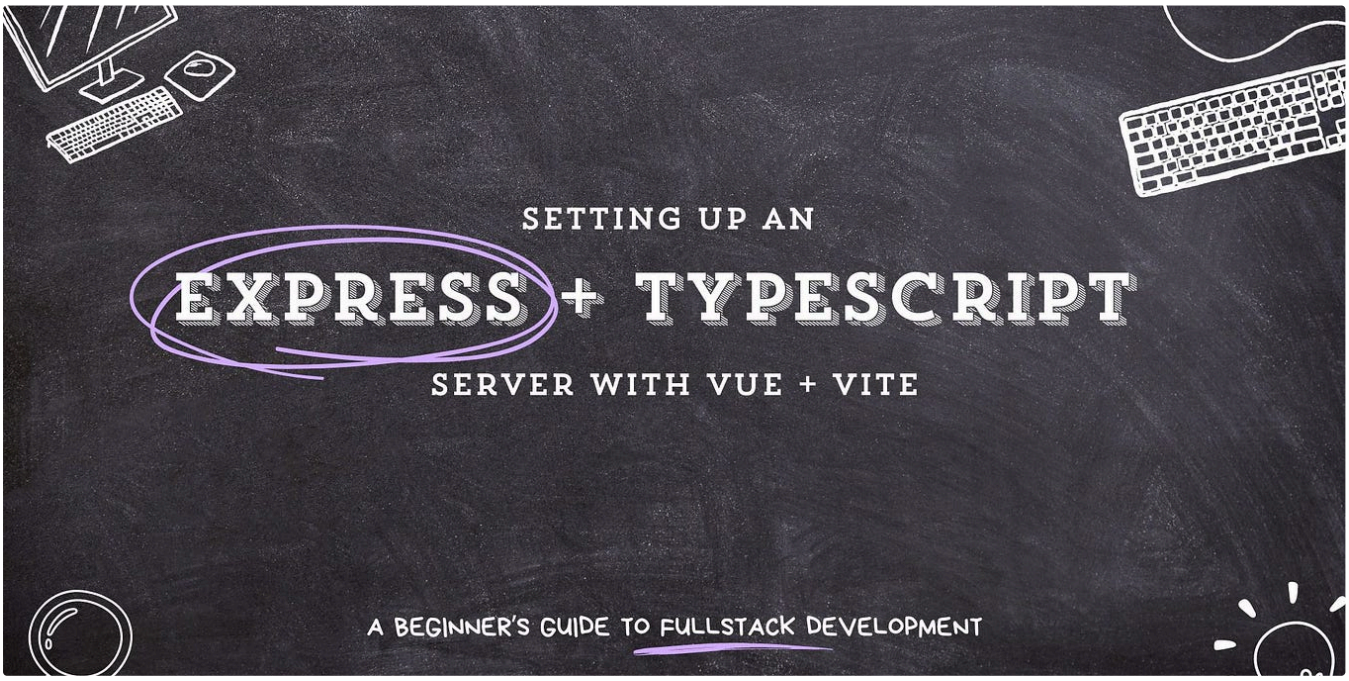
May 28 4

Muhammad Naqeeb

Building a Dockerized Node, Express, and MongoDB App with Docker Compose

Welcome to our blog on building a Dockerized Node.js, Express, and MongoDB application using Docker Compose! In this guide, we'll walk you...

Mar 21 1



Monique McIntyre

Setting up an Express + Typescript Server with Vue + Vite

Venturing into backend development can feel like stepping into uncharted territory, especially for us frontend developers. As a...

Mar 22 6 2

See more recommendations