**Next Documentation**

NextJS is a react framework that help us to make search engine optimized react app with 0 configuration. A traditional React app is rendered client side where the browser starts with a shell of an HTML page lacking any rendered content. From there, the browser fetches the JavaScript file containing the React code to render content to the page and make it interactive. But there are two major drawbacks with client-side rendering. One, the content is not reliably indexed by all search engines or read by social media link bots, and two, it can take longer to reach the first Concertful paint when a user first lands on the webpage. Next is a framework that allows you to build a React app but render the content in advance on the server. So, the first thing a user or a search bot sees is the fully rendered html. After receiving this initial page, client-side rendering takes over and it works just like a traditional React app. It's the best of both worlds, fully rendered content for bots, highly interactive content for users. Inside of a next project, you have a “pages” directory. Each JavaScript file defined here exports a React component that represents a route in the application. In other words, the file structure here mirrors the actual URLs that the user will navigate to and next provides its own router to make navigation seamless.

It supports,

1. Server-Side Rendering (SSR) -> ready food and self-cook food analogy.

2. Static Site Generation (SSG)

3. Built-in Router

4. And a lot of other tools for SEO of websites.

Essentially, NextJs has revolutionized the process of routing through its file-based routing, which we are going to learn about in the coming videos. Summarising, using NextJS helps you develop React websites in a very easy, convenient, and SEO-friendly way.

With the help of NextJs, you get a lot of options for rendering on your website. NextJs is a great tool for static websites with no backend connections. If you wish, NextJs will provide you API routes as well, so it is not simply a front-end framework but also a back-end framework in some way or the other. And that enables you to program both the front end and the back end in NextJs.

A typical react application is rendered on the client-side, i.e., the JavaScript bundle is loaded and then takes over to populate the DOM using ajax calls. And if you could somehow open the source code of some React application, you wouldn't find the page content there itself because the content, in most of the cases, gets populated later through JavaScript. Well, this counts as one big problem for social media websites and search engines as they cannot find content in the HTML rendered by the react app. Well, NextJs addresses all such shortcomings a simple React application has, simply by introducing static site generation and server-side rendering, which ships the populated DOM to the client's browser!

* Install NodeJS through the installer with all default settings. That may take some time. Wait while it finishes.
* And now, I hope you all have already downloaded and installed VS Code on your systems. Download and install now, if you haven’t already.
* I have created a folder named NextJs Course and this is where I’ll put all my files pertaining to the course.
* Open VsCode in the same folder.

Now, you might wonder if learning JavaScript or React is necessary to utilize NextJS, and my answer would be, yes! You do need to know the basics but that is what I’ll do in the beginning, and knowing things as and when required. We'll brush up on JavaScript basics as well as refresh our previous knowledge too.

Let’s now see how we develop Next Applications:

Next App Creation

Well, one can create individual files pertaining to Next applications too but we exploit a utility named Create Next App, and you can refer to the documentation Create Next App to have a knowledge of all of its uses. But that shouldn’t be a matter of worry for now since I'm here. There are methods we follow to create a Next app, but we solely stick to the yarn one for its advantages over npm. And that’s all we will cover in the coming videos while we slowly dive into the playlist.

So, NextJs is basically nothing but a framework for React. And you will develop React only, but with a little more convenience, and better and a lot more features. This concludes our discussion for today. Next.js is used in tens of thousands of production-facing websites and web applications, including many of the world's largest brands. I simply cannot hold my excitement while we have to wait for the next video where we’ll build our first Next application.

**Install yarn using this command:**

npm install --global yarn

Creating your first Next.js app

You can create your first Next.js application by running the following command:

yarn create next-app

This command will ask you a few basic questions (the name of your app, etc.) and will create a folder with your next.js app inside of it.

You can open the generated Next.js code in the editor of your choice. I am using VS Code.

**A glance into the generated code**

Create next app generates a bunch of useful files with a git repository initialized. You will also see some files some of which are discussed below:

lock – This is the yarn lock file which stores all the packages with version numbers installed inside the node\_modules

.gitignore – A file that stores what is ignored by the git version control system

public folder – Contains the public directory of our Next.js web application. Anything put inside this folder will be publicly accessible on the web application’s server. Developers should avoid putting secret files inside this folder as they can easily be accessed by public users!

pages folder – Pages (which are essentially React components) following the file-based routing approach (more on this in the later lessons),

styles folder – Contains the CSS styles for the Next.js web application

**Starting the server**

If you look into the package.json file, you will find a script to run the development server. Open your terminal and run the following command:

yarn run dev

This will start the development server. Open the URL http://localhost:3000 (the URL shown in the terminal) by pressing control and clicking on it.

**Next.Js FAQs**

**Is Nextjs a Frontend or a Backend Framework?**

Its both, it can generate a frontend bundle along with APIs which are run on the server.

**Is NextJs better than React?**

No, and Yes! React is raw and Next.js comes with a few more features on top of that like file-based routing, Server-Side Rendering (SSR), Static Site Generation (SSG), a built-in router, etc.!

**File-Based Routing**

While using react, we need to install a third-party package to get the routing

Next.js comes built-in with an incredible feature called file-based routing. This feature enables Nextjs developers to simply create files inside the pages folder and those files will serve as routes for the application. For example, if you create 3 files (about.js, contact.js, and index.js) inside the pages folder of your Next.js application, you can view these components by visiting *localhost:3000/about*, *localhost:3000/contact*, and *localhost:3000/* respectively.

This is what file-based routing is essentially about! When a file is added to the “pages” directory, it's automatically available as a route.  Each page exports a react component which is rendered when the corresponding URL is visited by the person visiting the page!

A sample about page (component) will look something like this:

function About() {

return <div>About</div>

}

export default About

Visiting *http://localhost:3000/about* will display “About” on the screen as this component returns a <div> with “About” inside of it!

**Nested Routes**

To create nested routes in Next.js, all we need to do is to create corresponding nested folders. For example, If you want to create a route at *localhost:3000/about/one* , all you need to do is to create a folder named “about” with a component named “one.js”

**Dynamic Routes**

Defining routes as shown in the Nested Routes section above is not always enough for complex applications. In Next.js you can add brackets to a page ([sno]) to create a dynamic route (or URL slugs).

Consider the following page *pages/blog/[sno].js*:

import { useRouter } from 'next/router'

const Post = () => {

const router = useRouter()

const { pid } = router.query

return <p>Post: {pid}</p>

}

export default Post

This page will render Post: 1 when you visit <http://localhost:3000/blog/1> and the same page will render Post: 34 when you visit <http://localhost:3000/blog/34>

**JavaScript**

So, I won't start from the beginning covering every single functionality, rather I will just cover some things I perceive you might get yourself caught up in. Async Await, Destructuring, Promises and Callback, Spread operator are a few of them. And since we have already downloaded Node Js, there is nothing more you would need for now. So, let’s just get started with them.

**1. Spread Operator**

Suppose we have a javascript array *arr*containing integersand we want to pass the contents of this array into a function *avg* which takes three numbers and returns their average.

The syntax for the spread operator is three dots(...). So, when we pass this array *arr* containing three elements into the function *avg*, using the spread operator, it gets dilated itself, and we get our desired output.

// Spread operator

arr = [1, 2, 4]

function avg(a, b, c) {

return (a + b + c) / 3

}

let a = avg(...arr)

console.log(a)

And you might wonder what happens if there were more numbers in the array. Well, any number of extra inputs gets ignored by the function. So, there isn’t anything to worry about.

Another interesting use case of the spread operator is that we can copy an existing array into an entirely new array. We can even concatenate the contents of two different arrays into another different or new array.

arr = [1, 2, 4, 6, 32]

let a = [4, 7, ...arr, 67]

//makes a = [4, 7, 1, 2, 4, 6, 32, 67]

let b = [...arr]

//makes b = [1, 2, 4, 6, 32]

d = [5, 7, 9]

let c = [...arr, ...d]

//makes c = [1, 2, 4, 6, 32, 5, 7, 9]

Let’s now see how the spread operator works on javascript objects. Suppose we have a javascript object *obj1* having names, *name1, class1 and favLang*and values*Harry, 5b and JavaScript*respectively. Now, we want to create a different object *obj2* where everything is the same as that of *obj1* except the fact that the *favLang* should be *Python*and not *JavaScript.*Well, that could also be done using a naive method, using the dot method, but the spread operator makes it a one-liner. We just have to feed *obj1* into *obj2* and write all that we want to change as we have done below.

let obj1 = {

name1: "Harry",

class1: "5b",

favLang: "JavaScript"

}

//naive method

// let obj2 = {}

// obj2.name1 = obj1.name;

// obj2.class1 = obj1.class;

// obj2.favLang = "Python";

//using spread operator

let obj2 = { ...obj1, favLang: "Python" }

**2. Destructuring**

Suppose we still have the JavaScript object *obj1* and we have three different variables named *name1, class1, favLang* and that we would like to give these variables the same values as those given in the object *obj1.*Again, the naive approach would be to do them individually, but JavaScript allows you to do the same thing in just a single line. You simply have to comma separate the variables in a curly bracket and feed them the object variable whose values they need to possess.

let obj1 = {

name1: "Harry",

class1: "5b",

favLang: "JavaScript"

}

//naive method

// let name1 = obj1.name;

// let class1 = obj1.class;

// let favLang = obj1.favLang;

//Destructuring

let { name1, class1, favLang } = obj1;

**3. Async Js**

JavaScript is asynchronous in nature. It is different from other programming languages where they have a blocking nature. The blocking nature refers to the fact that they don't execute a line until all lines preceding it has been executed. JavaScript doesn't actually wait for a line to get executed before moving on to the next one. Follow the snippet below:

// 1

console.log("Harry is a good boy!");

// 2

setTimeout(() => {

console.log("Harry is inside setTimeout");

}, 1000);

// 3

console.log("Harry is a bad boy!");

Here, any other programming language would have executed lines in order 1,2, and 3. But since line 2 demands a Timeout of 1000ms i.e. a second, JavaScript doesn't wait for that timeout to get finished rather it moves ahead, and executes line 3 first, and then line 2 after a second.

Harry is a good boy!

Harry is a bad boy!

Harry is inside setTimeout

**4. Promises and Callback**

We’ll use the file system module *fs*which is a module to read files in NodeJs. Using that we’ll read a file named *file.txt* encoded using *utf-8*. It takes some time to read a file, but that doesn't stop the program from executing lines that follow. Follow what the snippet says.

let fs = require("fs");

fs.readFile("file.txt", "utf-8", (err, data) => {

console.log(err, data);

})

console.log("This is eof")

Here, the readFile method just makes sure that *fs* reads the file *file.txt* using the utf-8 formatting, and calls the function as soon as it finishes reading the file. And while the program reads the file, it wouldn't sit idle rather execute lines that follow. And that is why the output has the second line printed first.

This is eof

null Harry is a good boy

So, this was a callback. Imagine how chaotic and unorganized things would get if there were multiple nested callbacks. And that is why we use promises.

Javascript file system module has a promise feature where the readFile method returns a promise and this state can even be stored. We can then use a dot then method to command the program about what it would do after the promise resolves or the file gets read.

// let fs = require("fs");

let fs = require("fs/promises");

let a = fs.readFile("file.txt", "utf-8");

a.then((data) => {

console.log(data);

})

console.log("This is eof")

And now, it would be comparatively much more readable than callbacks.

**5. Async - Await**

Consider the same file system module *fs* and let's make a function named *readThree*whose job would be to read three text files. The function takes three argument files, *file1, file2*and*file3*and reads them using the methods we discussed above. Now, it might be the case that file 2 gets read before file 1 or vice versa. So, it's pretty unpredictable.

And this is where we use the async await method which makes it possible for the program to follow the order we decide.

let fs = require("fs/promises");

const readThree = async (file1, file2, file3) => {

let a1 = fs.readFile("file1.txt", "utf-8");

let a2 = fs.readFile("file2.txt", "utf-8");

let a3 = fs.readFile("file3.txt", "utf-8");

let c1 = await a1;

console.log(c1);

let c2 = await a2;

console.log(c2);

let c3 = await a3;

console.log(c3);

}

And this is how you convert an asynchronous non- blocking program into a sort of blocking one if I were to put it straight and simple.

**6. JavaScript Map**

Javascript map is one of the most useful methods. It is primarily used to traverse every element of an array while calling a function on each of them. It doesn't change the original array in any way, rather returns a new array.

let a = [1, 2, 3, 4, 5];

let a2 = a.map((value) => {

return value \* 2;

})

// a2 = [2, 4, 6, 8, 10]

**7. Filter**

JavaScript filter is a javascript method to filter elements out of a container based on some parameter. Suppose there is an array of numbers and we want to create another array out of the elements of the first array having value greater than 6.

let a = [1, 2, 3, 4, 5, 64, 34, 12, 9];

let a2 = a.filter((value) => {

return value > 6;

})

// a2 = [64, 34, 12, 9]

**8. Reduce**

JavaScript reduce is another method that takes two elements of an array at a time. It applies any of the user-defined operators and returns a single value in the end which is the combined result of the whole array.

let a = [1, 2, 3, 4];

let a2 = a.reduce((val1, val2) => {

return val1 + val2;

})

// step1: 1+2, 3, 4

// step2: 3+3, 4

// step1: 6+4

// a2=10

So, here we added the whole array. It could have been any operator.

**9. Triple Equality**

Triple equality is a very simple concept. In contrast to other programming languages, where we compare two elements by using a double equality operator, JavaScript first converts both elements to the same data type before comparing them. With this, (1==“1”) return True. Therefore, we have a triple equality operator, which returns True only when the value and data type both are the same for two elements.

if (1 === "1") {

console.log("I am true")

}

else {

console.log("I am false")

}

//Outputs I am false.

That was enough JavaScript I believe for you to catch up on things. Let’s get started with the React basics.

**React**

It might already be clear to you that we put React components in the pages folder while NextJs is implemented. And that makes learning React an integral part of NextJs. Now the reason why React came into existence was to make user interfaces development easy.

React is a JavaScript library that is used to develop single-page applications. Single-page applications mean that applications developed using React do not require reloading to navigate between pages. This ensures a smooth user experience. React pages are built using JSX (JavaScript Syntax Extension), which is nothing but HTML with additional features that allow us to use JavaScript. React further lets you divide complex websites into simpler components. This saves us time when we need to change something for one particular section, and we can just make changes in one place.

We’ll now be working towards developing a simple application that would cover all the basics of React.

**Installation**

We have already downloaded Node Js. To check if it’s there on your system, use the command*npm -v* in your powershell.

**Creating React App:**

Choose the directory of your choice and open your PowerShell window where you want your app to get built. Now, simply run the following command in your terminal. Here, we have used the name *todos-list*as the name of our application. *Npx* is an *npm*package runner.

npx create-react-app todos-list

The installation of different useful packages might take some time. Once done, a folder named *todos-list* will appear there in the directory. Now, open that folder in VS Code. To give you a quick review of what those files are, there are two important folders, *Public* and *src*which contain an HTML file which is the only HTML file of the application and different utilities, css and js files respectively.

And now, we’ll start our development server by running the command,

npm start

This will redirect you to your browser where your application will start running at <http://localhost:3000/>. Having completed all the initials, we are now ready to move on to developing our project further.

App js is the file where the current JSX for the default page is written which is rendered through the index.js. App js has a function that returns a JSX template. Well, HTML and JSX have subtle differences, so be sure not to mix them up. JSX *className* corresponds to HTML *class*. You can write JavaScript in JSX only by using curly brackets. So, anything you return in those functions should be wrapped in another JSX element, be it empty tags.

Let’s now move further creating separate components for our website. But before that, you should have enabled the use of Bootstrap in your program. Just put these two codes in your index.html. The JS part should be put down below the body tag and the CSS part should be put in the head tag.

**JavaScript**

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0-beta3/dist/js/bootstrap.bundle.min.js" integrity="sha384-JEW9xMcG8R+pH31jmWH6WWP0WintQrMb4s7ZOdauHnUtxwoG2vI5DkLtS3qm9Ekf" crossorigin="anonymous"></script>

**CSS**

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0-beta3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-eOJMYsd53ii+scO/bJGFsiCZc+5NDVN2yr8+0RDqr0Ql0h+rP48ckxlpbzKgwra6" crossorigin="anonymous">

Now, we wouldn’t want everything to be developed in the same App component, rather we’ll use this App component to lead all the other components we’ll create starting with the Header Component.

**Creating the Header Component**

This component will contain the navigation bar of our React app. For convenience, we’ll put all our components inside a new folder named *MyComponents* inside the same src folder. Create a new file named *Header.js* inside that. Since the header component contains only the nav bar for our project, we will return the code for the navigation bar inside the react functional component. Below is what our header.js would look like after inserting the code for the navigation bar.

import React from 'react'

export default function Header(props) {

return (

<nav className="navbar navbar-expand-lg navbar-light bg-light">

<div className="container-fluid">

<a className="navbar-brand" href="#">{props.title}</a>

<button className="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarSupportedContent" aria-controls="navbarSupportedContent"

aria-expanded="false" aria-label="Toggle navigation">

<span className="navbar-toggler-icon"></span>

</button>

<div className="collapse navbar-collapse" id="navbarSupportedContent">

<ul className="navbar-nav me-auto mb-2 mb-lg-0">

<li className="nav-item">

<a className="nav-link active" aria-current="page" href="#">Home</a>

</li>

<li className="nav-item">

<a className="nav-link" href="#">About</a>

</li>

</ul>

<form className="d-flex">

<input className="form-control me-2" type="search" placeholder="Search" aria-label="Search" />

<button className="btn btn-outline-success" type="submit">Search</button>

</form>

</div>

</div>

</nav>

)

}

**Props** are nothing but parameters which are Javascript objects passed from the parent component to the child component. Import the Header component inside the App.js file since that is one main component. So, put the below code in the App.js file to import the header there.

Your App.js should look like the below code to render your Header file. We have used the title as a parameter that can be passed from App.js to Header.js.

import './App.css';

import Header from './MyComponents/Header'

function App() {

return (

<>

<Header title="ToDos List"/>

</>

);

}

export default App;

Well, what if we don’t pass anything for the prop object from the parent component. That particular prop will get a value undefined and for reducing that vulnerability, we use the **defaultProps** option.

The Default Props property sets the defaults values for props attributes if the parent component does not send values for the attributes. You simply have to put this code in your header file. This sets the default title value to *Your title here.*

Header.defaultProps = {

title: "Your title here"

}

Having done the Header component, we’ll move to complete our other components.

Next, we have the *Todo Component,*which will contain the list of all the *ToDo Items*.

Create another file named *ToDos* in *MyComponents* and don’t forget to import it in the main App component.

For now, we’ll just put some items ourselves in a JavaScript object named *todos.* Pass this object as a prop to the *ToDos* file. Your App.js must look like this now.

import './App.css';

import Header from './MyComponents/Header'

import { Todos } from './MyComponents/Todos'

function App() {

let todos = [

{

sno: 1,

Title: "Go to the market",

Desc: "You need to go to the market to buy some fresh vegetables"

},

{

sno: 2,

Title: "Go to the mall",

Desc: "You need to go to the mall to buy some gucci shoes"

},

{

sno: 3,

Title: "Go to the voting center",

Desc: "You need to go to the voting center to vote"

}

]

return (

<>

<Header title="ToDos List" />

<Todos todos={todos} />

</>

);

}

export default App;

Now, we need to create our Todo component. The Todo component has a job to display all the ToDo items, and as per what we have learned lately, we can use the JavaScript Map method to do the same. So, Todos.js has the below-mentioned code.

import React from 'react'

import { TodoItem } from "./TodoItem";

export const Todos = (props) => {

return (

<div className="container">

<h3 className="text-center my-3"> Todos List : </h3>

{props.todos.map((todo) => {

return <TodoItem todo={todo} />

})}

</div>

)

}

But for this to work, we must build our child component for Todos, the TodoItem component, whose only job is to display the item's content. So, create a new file named*TodoItem.js* in the same folder. Follow the snippet below.

import React from 'react'

export const TodoItem = ({ todo }) => {

return (

<div>

<h4>{todo.Title}</h4>

<p>{todo.Desc}</p>

</div>

)

}

This should display all the items we hardcoded. But now, we have to make it dynamic. We will add the buttons to add and delete the items from the list.

**Delete button**

Add a button element in the *TodoItem* file. For this to work, we’ll use another prop, named *onDelete* which will have a method *{onDelete}*. This method is originally created in the App file, passed onto as a prop to the *Todos file,*and from there to the *TodoItem*file. And there we use the onclick event to call the method with the parameter it demands. Carefully follow the changes we have done to the files to implement this delete button.

App.js file :

import './App.css';

import Header from './MyComponents/Header'

import { Todos } from './MyComponents/Todos'

import { TodoItem } from './MyComponents/TodoItem'

import React, { useState } from 'react';

function App() {

const onDelete = (todo) => {

setTodos(todos.filter((e) => {

return e !== todo;

}));

}

const [todos, setTodos] = useState([

{

sno: 1,

Title: "Go to the market",

Desc: "You need to go to the market to buy some fresh vegetables"

},

{

sno: 2,

Title: "Go to the mall",

Desc: "You need to go to the mall to buy some gucci shoes"

},

{

sno: 3,

Title: "Go to the voting center",

Desc: "You need to go to the voting center to vote"

},

]);

return (

<>

<Header title="ToDos List" />

<Todos todos={todos} onDelete={onDelete} />

</>

);

}

export default App;

Todos.js file:

import React from 'react'

import { TodoItem } from "./TodoItem";

export const Todos = (props) => {

let myStyle = {

minHeight: "70vh",

margin: "40px auto"

}

return (

<div className="container" style={myStyle}>

<h3 className="my-3">Todos List</h3>

{props.todos.length === 0 ? "No Todos to display" :

props.todos.map((todo) => {

console.log(todo.sno);

return (<TodoItem todo={todo} key={todo.sno} onDelete={props.onDelete} />

)

})

}

</div>

)

}

Copy

TodoItem.js file:

import React from 'react'

export const TodoItem = ({ todo, onDelete }) => {

return (

<>

<div>

<h4>{todo.title}</h4>

<p>{todo.desc}</p>

<button className="btn btn-sm btn-danger" onClick={() => { onDelete(todo) }}>Delete</button>

</div>

<hr />

</>

)

}

We used the useState hook to accomplish the job. Now, the delete button works all good. Next thing is to develop the add items functionality and for that we’ll create another file named *AddTodo.js.*AddTodo is the component where we have a form and the user enters the title and the description of the new job he wants to add to the list. Make sure you import them in the concerned files. Follow the code for the AddTodo file below.

import React, { useState } from 'react';

export const AddTodo = ({ addTodo }) => {

const [title, setTitle] = useState("");

const [desc, setDesc] = useState("");

const submit = (e) => {

e.preventDefault();

if (!title || !desc) {

alert("Title or Description cannot be blank");

}

else {

addTodo(title, desc);

setTitle("");

setDesc("");

}

}

return (

<div className="container my-3">

<h3>Add a Todo</h3>

<form onSubmit={submit}>

<div className="mb-3">

<label htmlFor="title" className="form-label">Todo Title</label>

<input type="text" value={title} onChange={(e) => setTitle(e.target.value)} className="form-control" id="title" aria-describedby="emailHelp" />

</div>

<div className="mb-3">

<label htmlFor="desc" className="form-label">Todo Description</label>

<input type="text" value={desc} onChange={(e) => setDesc(e.target.value)} className="form-control" id="desc" />

</div>

<button type="submit" className="btn btn-sm btn-success">Add Todo</button>

</form>

</div>

)

}

Submitting the form calls for the submit method which checks if the content is there, and if it is, it calls the method addTodo with the data fed to it as parameters. This addTodo is a function originally created in App.js. To update the todo list, we simply use the spread operator we learnt recently. Follow the final code for App.js below.

import './App.css';

import Header from "./MyComponents/Header";

import { Todos } from "./MyComponents/Todos";

import { AddTodo } from "./MyComponents/AddTodo";

import { About } from "./MyComponents/About";

import React, { useState, useEffect } from 'react';

function App() {

let initTodo;

if (localStorage.getItem("todos") === null) {

initTodo = [];

}

else {

initTodo = JSON.parse(localStorage.getItem("todos"));

}

const onDelete = (todo) => {

console.log("I am ondelete of todo", todo);

// Deleting this way in react does not work

// let index = todos.indexOf(todo);

// todos.splice(index, 1);

setTodos(todos.filter((e) => {

return e !== todo;

}));

console.log("deleted", todos)

localStorage.setItem("todos", JSON.stringify(todos));

}

const addTodo = (title, desc) => {

console.log("I am adding this todo", title, desc)

let sno;

if (todos.length === 0) {

sno = 0;

}

else {

sno = todos[todos.length - 1].sno + 1;

}

const myTodo = {

sno: sno,

title: title,

desc: desc,

}

setTodos([...todos, myTodo]);

console.log(myTodo);

}

const [todos, setTodos] = useState(initTodo);

useEffect(() => {

localStorage.setItem("todos", JSON.stringify(todos));

}, [todos])

return (

<>

<Header title="ToDos List" />

<AddTodo addTodo={addTodo} />

<Todos todos={todos} onDelete={onDelete} />

</>

);

}

export default App;

**Key changes we made:**

1. We checked if the serial number is currently at least 0. Otherwise, we directly set the serial number for the new entry to 0.
2. We didn’t hardcode some elements in the list ourselves. We connected it to the local storage. **LocalStorage** is a storage object to store the data on the computer locally. We save our content locally, converting them into JSON.
3. We used the useEffect hook to save our data in the local storage after any change was made.

**Using React Router:**

Now, this was just a page of our website. We want to have zero reloading for any inter-page transition. So that if we have another page, say named *about*, we could render it in between the same header and footer without any need for reloading. For that, install react-router-dom using npm.

npm install react-router-dom@5.2.0

Follow what we did to the JSX part of the App.js file.

return (

<>

<Router>

<Header title="My Todos List" searchBar={false} />

<Switch>

<Route exact path="/" render={() => {

return (

<>

<AddTodo addTodo={addTodo} />

<Todos todos={todos} onDelete={onDelete} />

</>)

}}>

</Route>

<Route exact path="/about">

<About />

</Route>

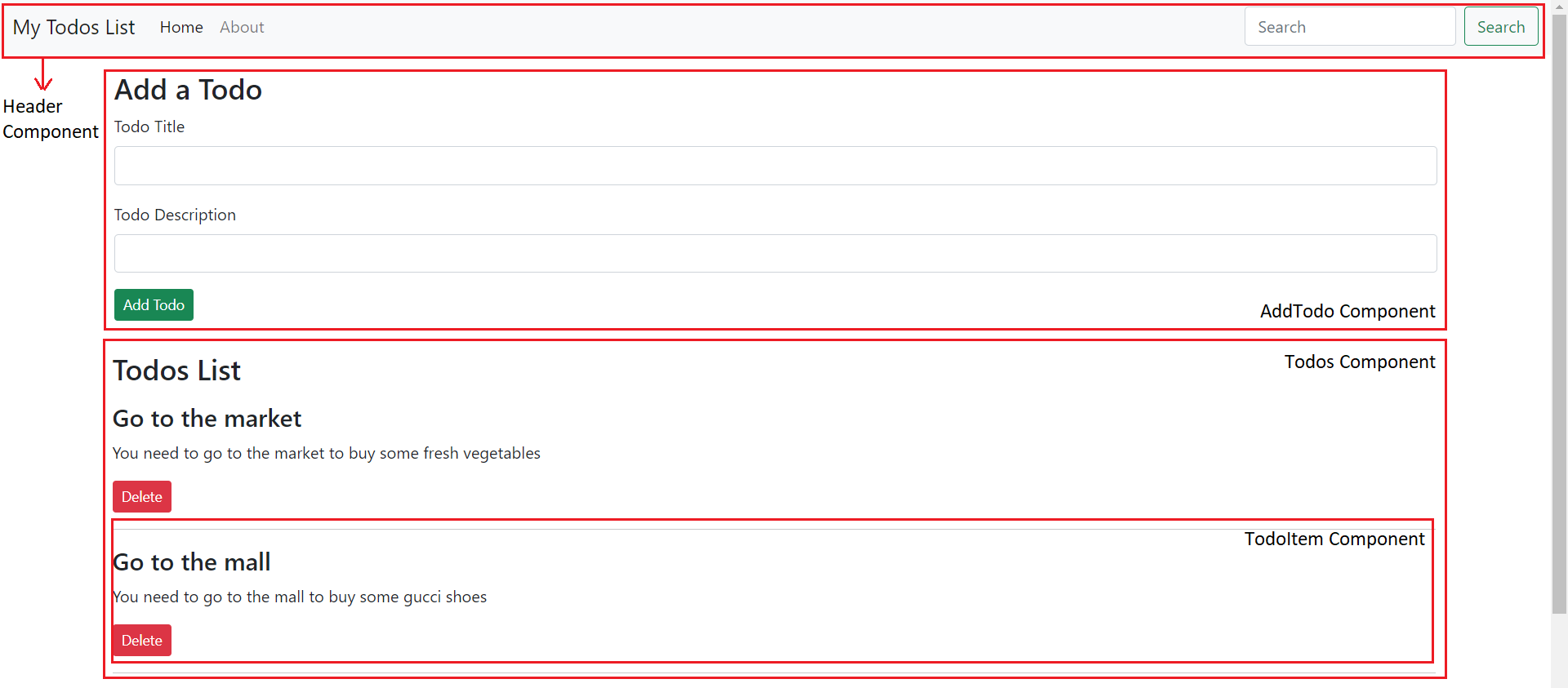
</Switch>

</Router>

</>

);

The final display of the webpage:



And here, we finish creating a sample project. Hopefully, this has given you a good idea of how JavaScript and React work.

# Head and Script Component in Next.js | NextJs Tutorial for Beginners #6

Regardless of whichever framework or library you use, the basics and essence of HTML, CSS and JavaScript remains intact well within your needs. You have the clarity of their applications and you know that the Head tag contains the title and the styling for the page but that is when you use core HTML. There might be times when you want to write stuff in the Head tag, but find yourself stuck in Next Js where there are pages which export React components which we write and there is no Head.

For your sake, let me tell you that Next Js can do wonders for you. Next Js provides a Head component which lets you inject stuff in Head and change things like title of the page directly from the title tag present in the Head component. Due to this very ability of Next Js, we can enjoy the new, great features of Next Js while still using the good old HTML, CSS, and JavaScript.

Let’s now see how it actually works.

## next/Head

As a part of the previous tutorial, we started creating a blogging website named Hunting Coder. We created the applications with a very minimal change in its content. Open the same folder where you created the hunting coder application.

If you could carefully observe the default template of the application and visit the index.js page, you would find an import named Head from next/head. If that import were assumed to be removed, you would get an error to see. And this is because the default template of the application uses the Head component and features an executable program in it.

<Head>

<title>Hunting Coder</title>

<meta name="description" content="Generated by create next app" />

<link rel="icon" href="/favicon.ico" />

</Head>

The Head component contains the title tag which for our website has Hunting Coder written in it and this gets reflected on the browser’s title bar.  We could have used JavaScript instead of Next if we had been using React, but Next gives us the convenience of injecting elements directly into the Head. Using this, you can add a custom title, CSS and any meta tags within the head component of your web page as per your requirements.

Some meta tags popularly used are,

* Description - This meta tag is used to define the description of the webpage.
* Keywords - This meta tag is used to define keywords for search engines.

The first thing was this. The other thing is that we can even inject external scripts into our programs by including them in the Head component, whether it is for Google Analytics or Search Console or anything. This is done using the <script> tag inside the <Head> tag. Script tag has a prop named src which gives it the location to the JavaScript file. Due to the time, it often takes for JavaScript to be executed, embedding external scripts into the program could sometimes cause trouble with how fast the webpage loads. Moreover, you do not want your website to slow down just because third-party external scripts failed to load on time. So, to overcome this, NextJs features a Script component as well.

## next/Script

This Script component could easily be included into our programs using the following import.

import Script from 'next/script'

Copy

A Script component operates as follows:

<Script src=''>

</Script>

Now this Script component features several attributes and one of them being the strategy. This is used to set the loading priority of external scripts improving the loading performance of websites. By default, strategy is set to afterInteractive which means the scripts are loaded immediately after the page becomes interactive. But for heavy third-party scripts, we prefer lazyOnload. This enables the scripts to get loaded only after everything else is rendered already.

<Script src=' ' strategy='lazyOnload'>

</Script>

And that's how using scripts inside Head differs in functionalities from using a separate Script component. With this, we have configured both our Head and our Script components.

**IMAGE Component**

At times, when you use HTML, CSS, or JavaScript or any other framework, it can happen that when you embed an image into your programs, you complain about it taking forever to load on the website because of the size of the image. As a result, your site runs slower and your search engine optimization is ruined.

As for Next Js, it has improvised on this front too. Next Js provides an Image component. Image component automatically optimises your images. By default, images in this component are lazy loaded. As a result, the loading time of the website is greatly reduced since only images in the viewport - that is, on the screen - get rendered and not those that are not visible to you. It is through tools like these that Next Js is strengthened. Now let's take a closer look at the Image component.

**Next Js Image Component**

Open the same folder where you have created the *hunting coder*application. The default template of the application contains an import named Image from next/image in the index.js file similar to what we studied in the Head component.

import Image from 'next/image'

Now, let me tell you what this Image component does.

1. It provides you with optimised images.
2. By default, images are lazy loaded which means that images which are not visible to you or are not there in your viewport would not get loaded.
3. Helps you make your images responsive conveniently.

There are a number of attributes this Image component comes with. We’ll discuss some of the most used and important ones. So, let’s start with running the development server.

yarn dev

You must find a default card set on the website. I’ll remove them first since I don’t need them to be there on my website. I’ll start by creating a <div> element having className *blogs.*This holds my list of blogs. Inside this <div> element, there is a heading named *Popular Blogs.*create another <div> element having className *blogItem.*This holds my individual blog post. Each blog would contain a heading <h1> and a paragraph <p> associated with it. So, the structure of the blog page would be something like this.

<div className="blogs">

<h2>Popular Blogs</h2>

<div className="blogItem">

<h3>How to learn JavaScript in 2022?</h3>

<p>JavaScript is the language used to design logic for the web</p>

</div>

<div className="blogItem">

<h3>How to learn JavaScript in 2022?</h3>

<p>JavaScript is the language used to design logic for the web</p>

</div>

<div className="blogItem">

<h3>How to learn JavaScript in 2022?</h3>

<p>JavaScript is the language used to design logic for the web</p>

</div>

</div>

Each of these blog items would get displayed using a loop. We will add our css in some later part of the tutorial. For now, we are planning out things we want to have on our website. It would also be handy to have a navigation bar on our website for convenience in browsing pages. So, let’s just get ourselves a nav component which is very easy to integrate.

**Adding a Nav Component**

First of all, add a <nav> element in the*index.js* file. Add a <ul> tag with for list elements inside the <li> tags namely *Home, About, Blog*and *Contact.*Here’s what the code looks like.

<nav>

<ul>

<li>Home</li>

<li>About</li>

<li>Blog</li>

<li>Contact</li>

</ul>

</nav>

But this surely doesn't appeal to us as a navigation bar. We would need to define a CSS styling for it to look like one.

**Styling the nav component**

Give a className to the nav component, say, *mainnav.*So, this goes as *{style.mainnav}*Since we don't want every similar component to have the same styling, defining a className is helpful.

Now move on to the main CSS file *Home.module*and add the CSS snippet provided below.

.mainnav ul{

display: flex;

justify-content: center;

margin-top: 50px;

}

.mainnav ul li{

margin: 0 24px;

list-style: none;

font-weight: bold;

}

We have changed our navigation component to a flexbox and with some proper coherent margins and paddings, it has come out well. Update the styling of the main component to the one given below.

.main {

min-height: 100vh;

flex: 1;

display: flex;

flex-direction: column;

justify-content: center;

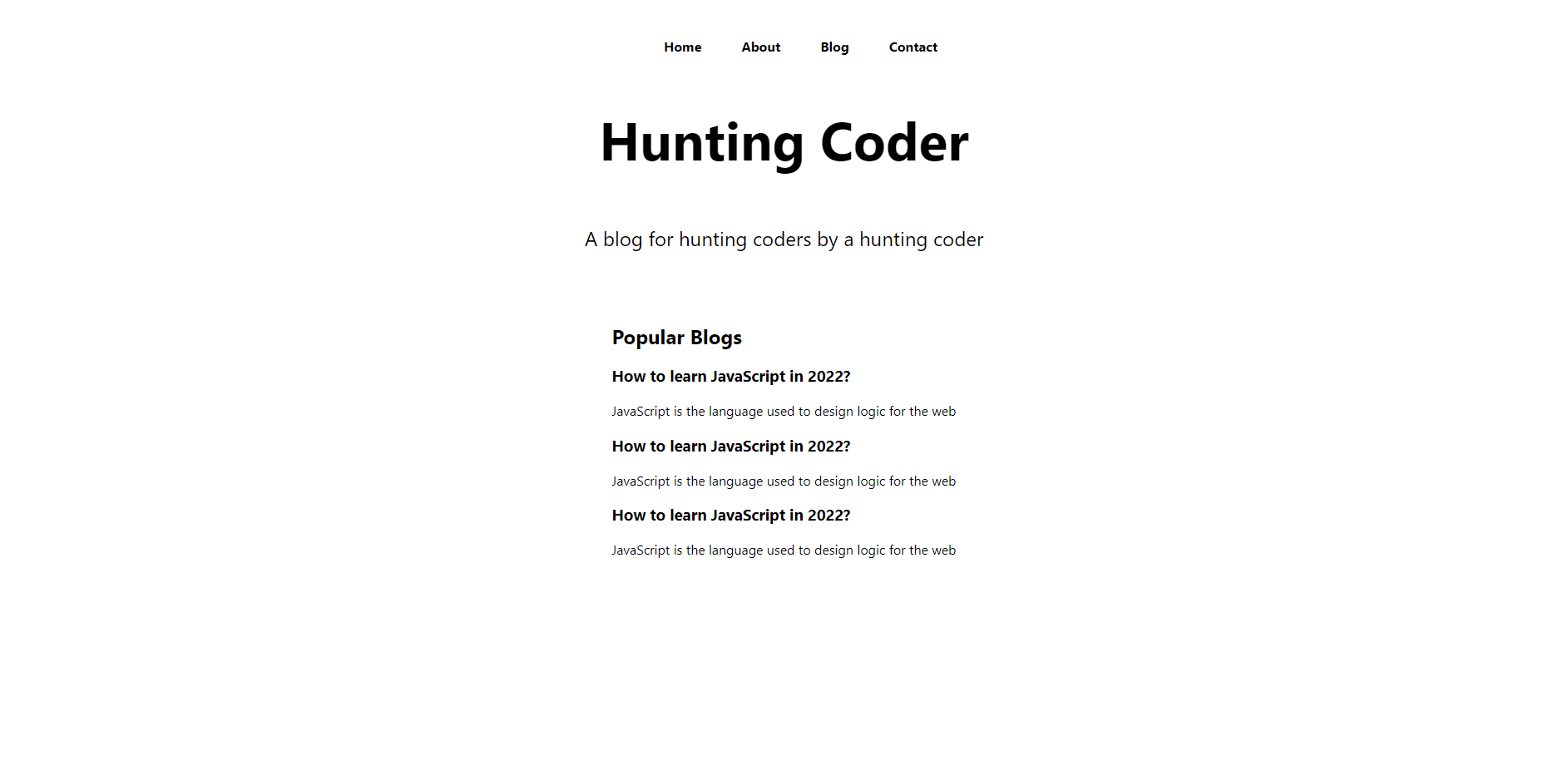
align-items: center;

justify-content: flex-start;

padding-top: 3rem;

}

We're now ready to proceed. This was our navigation component. We could have done more styling of the components but that is not our main focus here for now. We have done just enough of them to proceed. The final home page somewhat looks like this.



This was all for today. In the next tutorial, we might give a better structure to the whole website. We’ll also build different pages we would render from the navigation bar. We’ll make it more dynamic.

**Basic Dynamic Routing:**

[slug].js:

import React from 'react'

import useRouter from 'next/router'

const [slug] = () => {

    const router = useRouter();

    const {slug} = router.query;

    //Basic dynamic routing.

  return (

    <div>{slug}</div>

  )

}

export default [slug]

**Link Component**

When you use anchor tags to link your website pages, clicking the website reloads the page, whereas using Link for the same purpose avoids the triggering of the page refresh. And this is because there is no *href*but JavaScript is populating the content of the linked page. So basically, that’s the beauty of the Link component of Next Js. It prevents the web page from reloading each time you click a navigation link.

Link is a component which can be easily imported from next/link using the command given below.

import Link from 'next/link'

Now, since we want the contents of our navigation bar to handle links to different pages, we would nest the list items into the Next Js Link component. Link component has an only required attribute named *href* which holds the relative link to the page you want to navigate from this Link. We have placed these list items inside an anchor tag to make them appear as links. This is how the updated code looks like.

<ul>

<Link href='/'><a><li>Home</li></a></Link>

<Link href='/about'><a><li>About</li></a></Link>

<Link href='/blog'><a><li>Blog</li></a></Link>

<Link href='/contact'><a><li>Contact</li></a></Link>

</ul>

And as you can observe, we can now navigate to different pages through these links.

The Link component also has other useful and complex attributes, such as scroll, which lets us decide whether to scroll the page all the way to the top or not while navigating. More information about the Link component can be found on its official Next Js documentation page

Link accepts the following props:

* href - The path or URL to navigate to. This is the only required prop. It can also be an object, see [example here](https://nextjs.org/docs/api-reference/next/link#with-url-object).
* as - Optional decorator for the path that will be shown in the browser URL bar.
* prefetch - Prefetch the page in the background. Defaults to true. Any <Link /> that is in the viewport (initially or through scroll) will be preloaded. Prefetch can be disabled by passing prefetch={false}. When prefetch is set to false, prefetching will still occur on hover. Pages using [Static Generation](https://nextjs.org/docs/basic-features/data-fetching/get-static-props) will preload JSON files with the data for faster page transitions. Prefetching is only enabled in production.
* [replace](https://nextjs.org/docs/api-reference/next/link#replace-the-url-instead-of-push) - Replace the current history state instead of adding a new url into the stack. Defaults to false
* [scroll](https://nextjs.org/docs/api-reference/next/link#disable-scrolling-to-the-top-of-the-page) - Scroll to the top of the page after a navigation. Defaults to true

# **Adding Global Styles in Next.js**

In the last lecture, we saw how easy Next Js makes it for us to use different stylings at different components. We learnt component level CSS application and the functioning of it in detail. We saw the applications of modules and how ending any CSS file with .module.css could help us using them as modules.

And as you may remember, for every Next Js application that we create, there is an in-built CSS module. And classes present in those modules only are used in the default home page of the application. But Next Js gives us the flexibility to define our own CSS and import it into any component we wish to use them in. Today we'll look at how we can structure our CSS part for the same website, Hunting Coder and how the classes involved with those CSS files can be applied efficiently.

In addition, we will learn how to include a custom CSS file as we did before with HTML stylesheets. So, let's first create a CSS file named style.css in the styles folder. There won't be anything complex in this but a background style for demonstration purposes. Make sure you have run the development server already using yarn dev.

body{

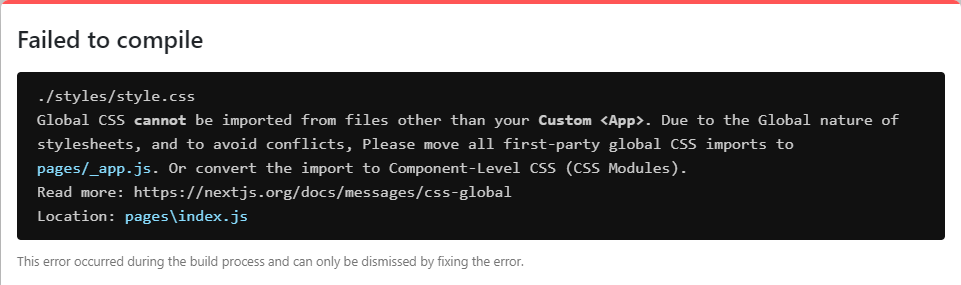
background-color: red;

}

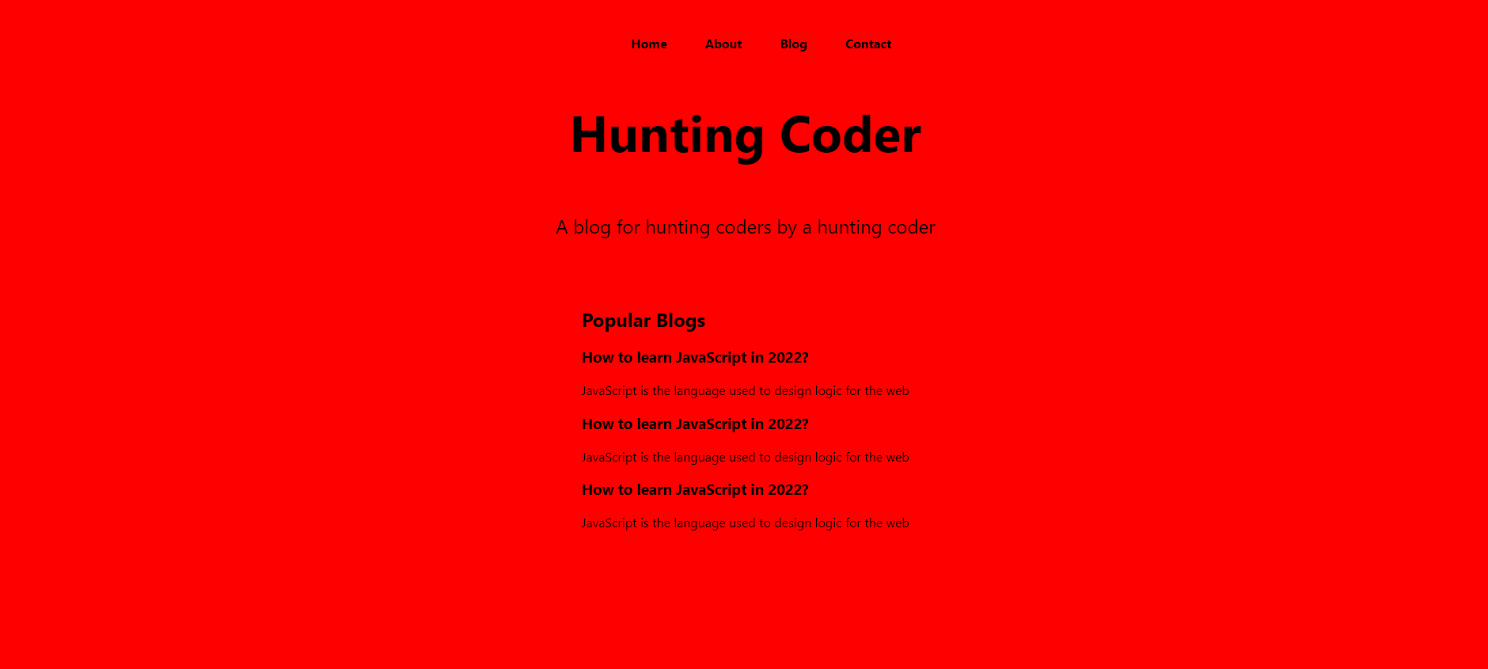
**Code Snippet 1: CSS in style.css**

</AD>

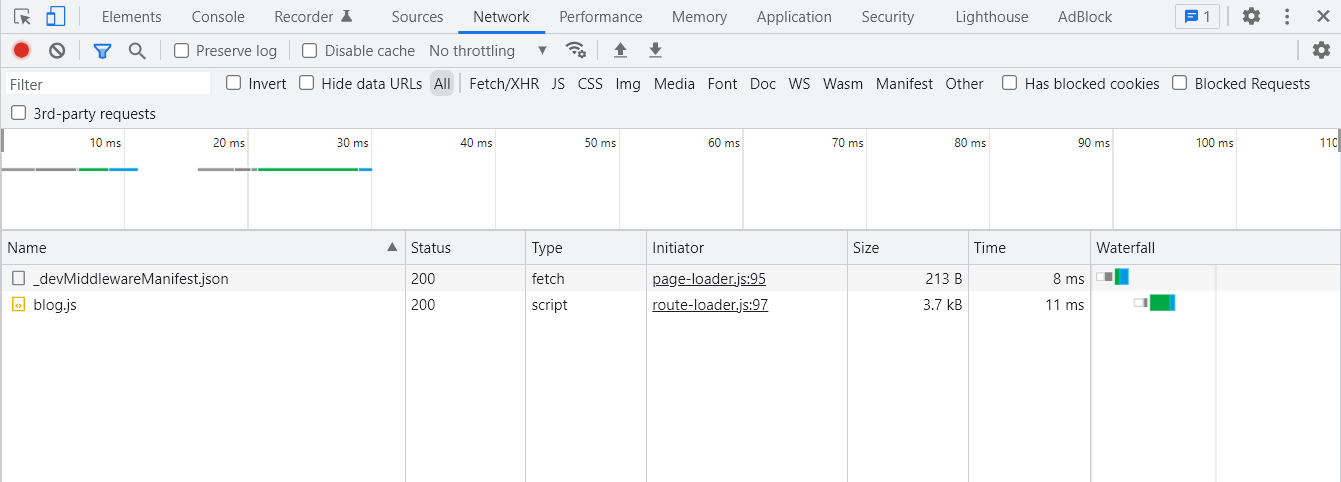
Now, we want to include this stylesheet into our website and that too in our index.js file. Earlier we learnt including the .module.css file using the import method and styles object. But the disappointing part is, you cannot import a stylesheet directly into your applications. In fact, even if you forcefully put the import command, you would get an error suggesting that you use the modules method to style your website.



The error asks you to either shift all CSS imports to the pages/\_app.js  file which makes all your custom stylesheets global or simply use the Component level CSS styling. So, let’s just put this import command in the \_app.js file. You would observe the changes took effect and our stylesheet has been included as soon as we followed the above command and this is because \_app.js is the file which renders all your website pages. Later, we will talk about \_app.js more in depth.

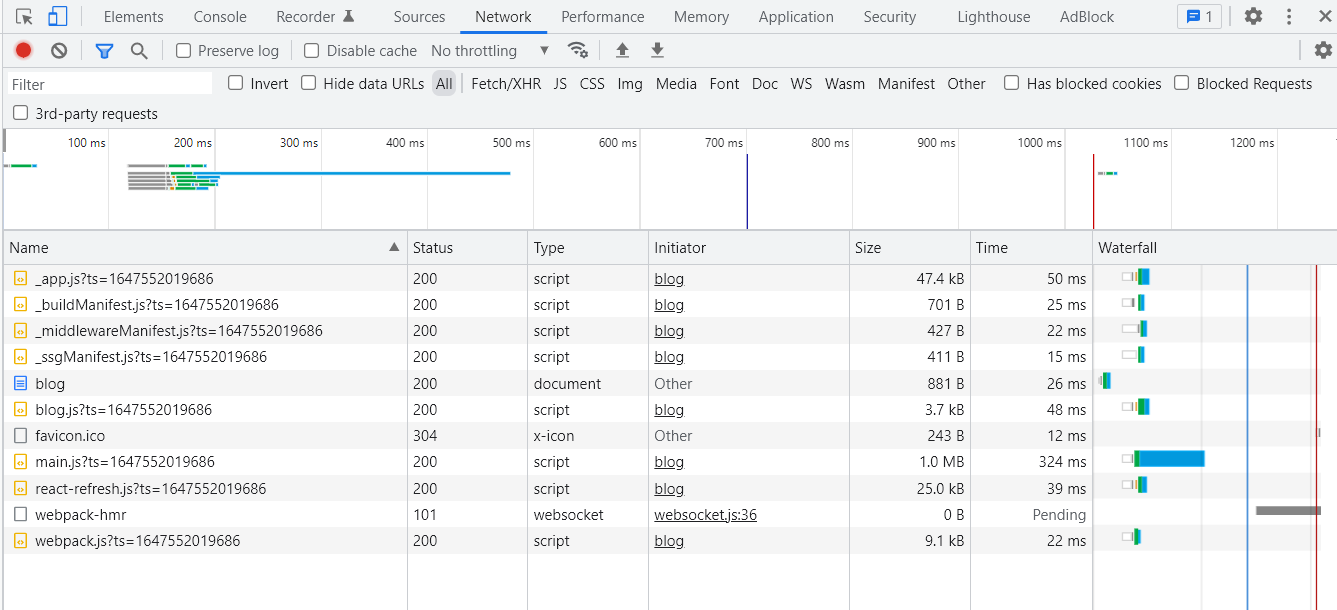


Furthermore, you can observe that the website does not take any time to load when you navigate between multiple pages. To provide a more concrete understanding, when you access your blog page from the home page, you can see what all things your website loads under your Networks tab. I have attached the screenshot of the same blow.



**Figure 1: Network tabs when we navigate from home to blog page**

It is evident that there are only a few things that are loaded here, each having a size of a few kilobits. This is what makes our pages load so quickly. Reloading the blog page now will show you the difference in the data being loaded now compared to when you navigated the page earlier. The size of the data rendered has become too big now.



**Figure 2: Network tabs when we reload the blog page**

Hope you understand the differences that Next JS provides over the way we used to create websites earlier. You may not have understood everything we discussed today regarding how CSS is used within Next JS and for someone just beginning, it's completely understandable. Eventually, you will get the hang of it. We will continue to revise things at times.

To sum it up, whenever you wish to apply Component level styling, you use the .module.css extension for your CSS files and import them directly and when you wish to define a global styling, you put import your CSS file in \_app/js file.

# **Styled jsx in Next.js**

Despite how great these technologies are, you'll always miss the simplicity of our good old HTML, CSS, and JavaScript. But Next JS lets you enjoy that convenience here as well. You can program your stylesheets targeting your classes just like you did in normal CSS even in Next Js with much more functionalities using a special syntax named **style JSX**. Using this makes it possible to write CSS specific to a particular component. That even makes it easier to add, change, and delete styles without worrying about how they will affect other components. You can even declare it globally following different methods. Let's just begin with style JSX without further ado.

First of all, run your development server using yarn dev. So far, we have discussed a lot about styling our web pages. Now, style JSX is another method to add CSS in a particular component. Suppose we consider the div element we are returning in index.js. Here, we define a style jsx tag, which is wrapped in the main div element. Inside it, we target a particular className, here mySpan, and give custom styling to it and everything inside this style jsx tag is wrapped in backticks.  Refer to the snippet below.

<style jsx>

{

`

.mySpan{

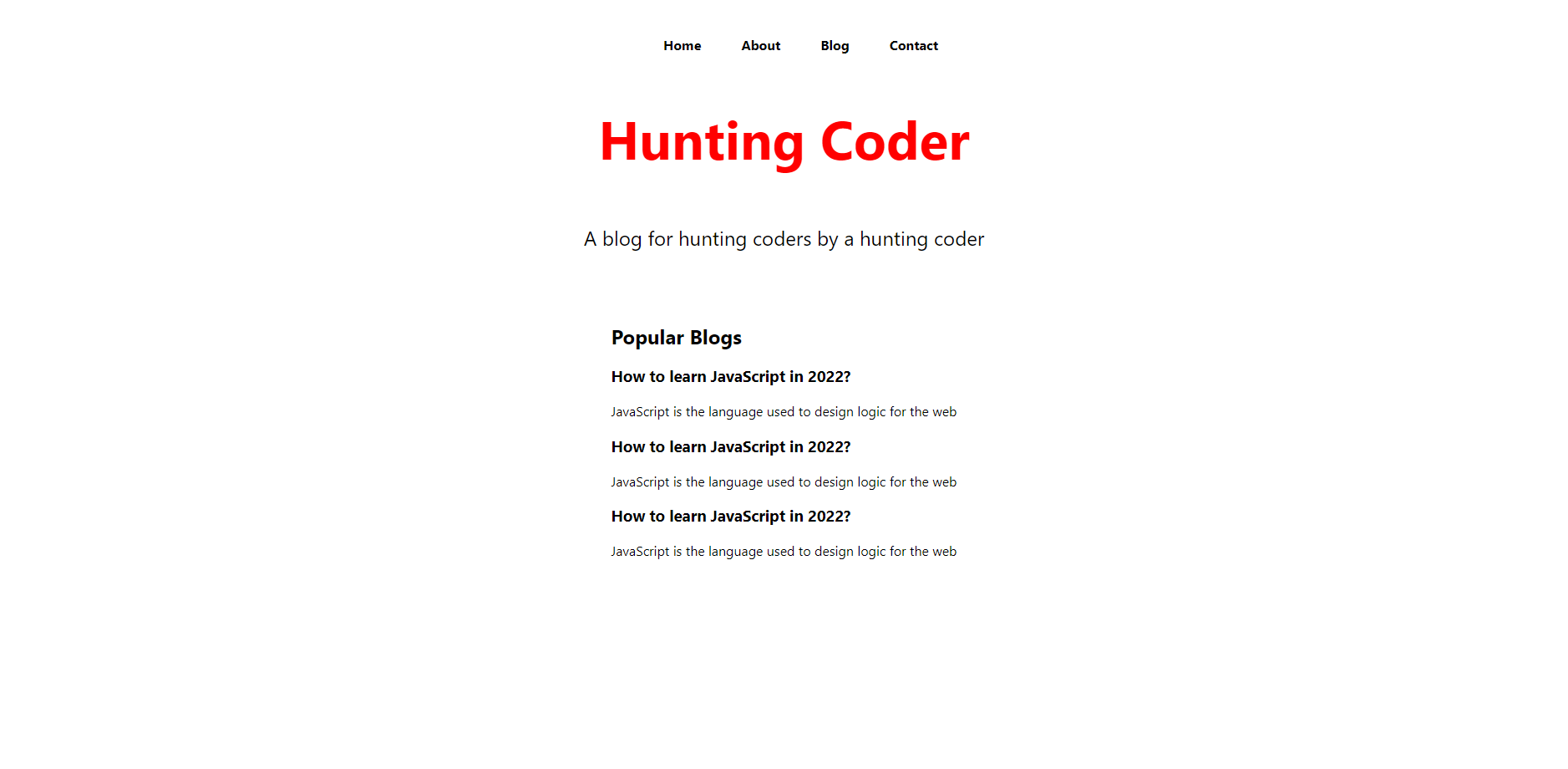
color: red

}`

}

</style>

We have given className mySpan to the heading of the webpage Hunting Coder which is wrapped in a span tag. Using the above styling method, we have applied the font colour red to the heading. This change takes effect once the program is saved.



In turn, this enables us to implement Component-level CSS the same way we did before. For you to see how that works, let’s first create a folder named components. Create a JavaScript file named dummy since this is just for demonstration purposes. Inside the file, define a react arrow function export component, shortcut for which is rface. Make sure the function name Dummy begins with an uppercase letter. Now, define a style jsx  in the return statement of this function as we did above, but this time the target className would be dummy. Set the background colour for this class to yellow. Define a div element inside the return statement for this component, and give it a className dummy. This is the element we are giving the above styling to.

import React from 'react'

const Dummy = () => {

return (

<>

<style jsx>

{`

.dummy{

background: yellow;

}

`}

</style>

<div className='dummy'>I am dummy dummy</div>

</>

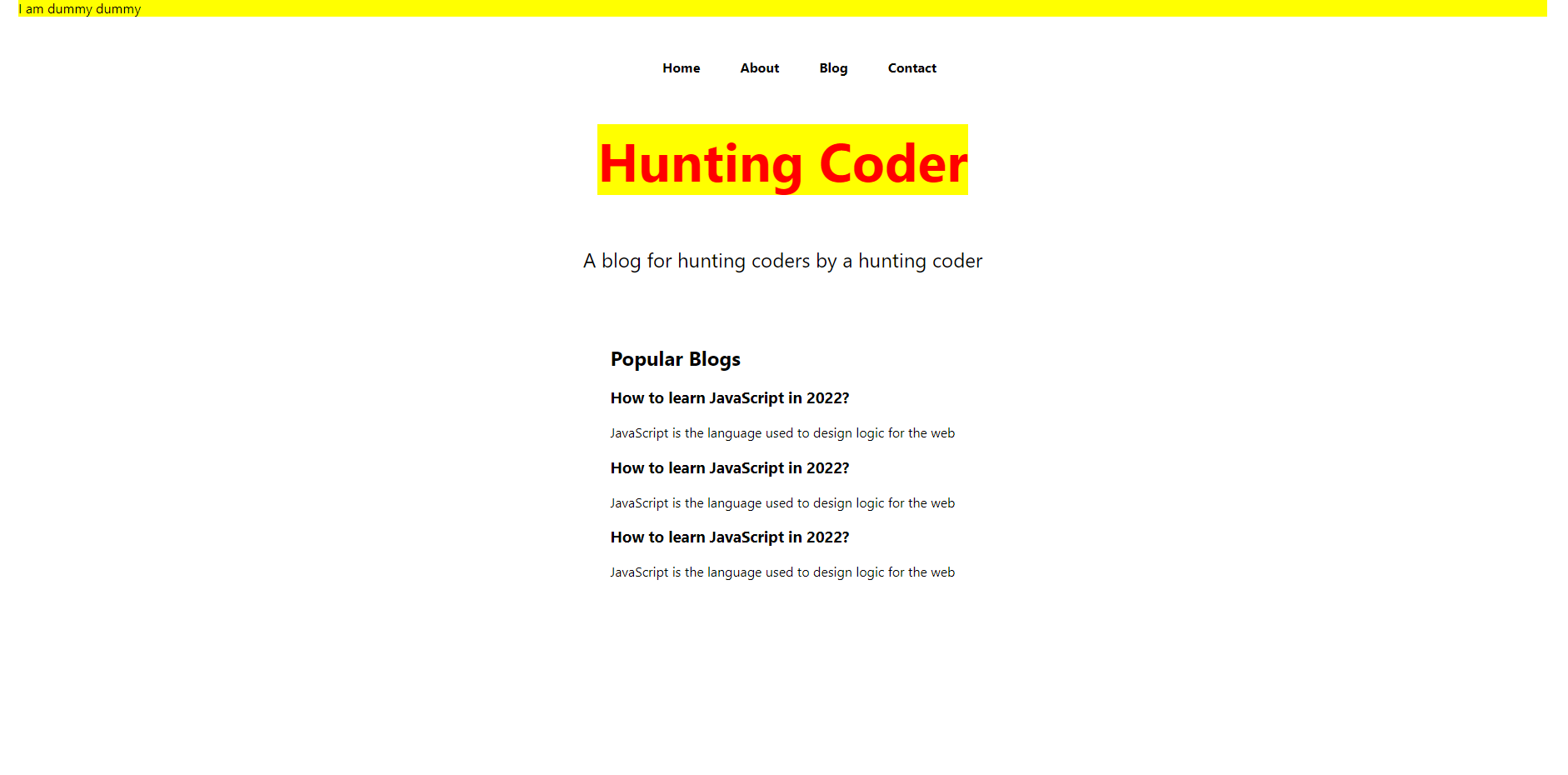
)

}

export default Dummy

rafce

The above implementation produced a very obvious result, as shown below.



To sum it up, whenever you wish to apply Component level styling, you use the .module.css extension for your CSS files and import them directly and when you wish to define a global styling, you put import your CSS file in \_app/js file.

**//BASIC Clean up of the proejct**

Currently, the navigation bar is only visible on the homepage of the website. As we navigate to other pages, such as the about page or the blog page, we only see demo texts and there is no navigation bar present there.

One way to achieve this would be to copy the navigation bar on those pages and change things accordingly. But, why does this process need to be tedious and redundant when we have the feature of components and we can exploit the *\_app.js*file to our benefit? In this lesson, we'll see how we can implement a navigation bar across all pages efficiently.

Run the development server using *yarn dev* from your main directory where your *Hunting Coder* folder is located. Now, integrating the navigation bar into all pages of our website is not that difficult, but the way we will implement it will be very useful for further applications down the road. And the file *\_app.js* is of utmost importance to us. This file can be found in the *pages* folder, and you might be wondering what it looks like. Well, *\_app.js* is different from any other page in that folder. It is the only page that gets rendered on every other page in the folder. This can be verified by putting a console log command somewhere in the *MyApp* function of the *\_app.js* file. And then you could observe that every time a page is loaded, this log will appear in the console.

For now, the *MyApp*function returns only a component and its corresponding *props*. However, we would simply wrap this part of the return statement inside an empty tag, so that we can now return any other piece of information as well. And this additional piece of information could be anything ranging from a simple *div* element to an entire component. This is what we will exploit to add a navigation component to every page.

So, we'll begin by creating a new component named *Navbar.js*so that we wouldn't have to include the whole snippet in the return statement. Construct an arrow function export component using the shortcut *rface*named *Navbar.*Cut the *nav*component from the *index.js*file and paste it here in this *Navbar*component. Do necessary imports such as *styles*and *Link.* This creates a separate *Navbar*component. We just have to include this now in the *MyApp*function in the *\_app.js*file.

In the *\_app.js*file, import *Navbar* from its relative location. Add *<Navbar/>*in the return statement of the *MyApp*function. And we are done. The navigation bar gets installed on every page. You can see it for yourself.

import React from 'react'

import styles from '../styles/Home.module.css'

import Link from 'next/link'

const Navbar = () => {

return (

<nav className={styles.mainnav}>

<ul>

<Link href='/'><a><li>Home</li></a></Link>

<Link href='/about'><a><li>About</li></a></Link>

<Link href='/blog'><a><li>Blog</li></a></Link>

<Link href='/contact'><a><li>Contact</li></a></Link>

</ul>

</nav>

)

}

export default Navbar

**Code Snippet 1: *Navbar.js*in the components folder**

import Navbar from '../components/Navbar'

import '../styles/globals.css'

function MyApp({ Component, pageProps }) {

return (

<>

<Navbar />

<Component {...pageProps} />

</>

)

}

export default MyApp

# **Creating Blog and Blogpost page | NextJs Tutorial for Beginners #15**

In the last lecture, we learnt how the \_app.js file works and how important it is in terms of executing something globally. We took advantage of this feature and made the navigation bar available on every page of the website by simply creating a separate Navbar component and including it in the return statement of the MyApp function in the \_app.js file.

Hunting Coder is a blogging website and the primary product of our website is the content in a blog. And this content gets displayed on our blog post page which we will create in this tutorial. In addition, we will create a blog page for listing all the blogs present on our website. Our preferences will guide the design of these pages. We’ll first start with the making of the blog page followed by the blog post page that can be generated dynamically by itself for each blog on the site.

Open the Hunting Coder folder in VSCode and start the development server using yarn dev. Once the server starts, you can see the navigation bar on each of the pages. While we begin the process of structuring our blog page, it is important to keep in mind that the blog page will be inspired by the styling of the list of blogs on the homepage.

### Creating the Blog Page

Open the blog.js file. Remove everything that is already present in the return statement of the arrow function Blog. Copy the section handling with the list of blogs on the homepage and paste them here inside the return statement. The blog.js file would have the following content for now.

import React from 'react';

const Blog = () => {

return <div>

{/\* <div className={`${styles1.con} ${styles2.con}`}> \*/}

<h2>Popular Blogs</h2>

<div className="blogItem">

<h3>How to learn JavaScript in 2022?</h3>

<p>JavaScript is the language used to design logic for the web</p>

</div>

<div className="blogItem">

<h3>How to learn JavaScript in 2022?</h3>

<p>JavaScript is the language used to design logic for the web</p>

</div>

<div className="blogItem">

<h3>How to learn JavaScript in 2022?</h3>

<p>JavaScript is the language used to design logic for the web</p>

</div>

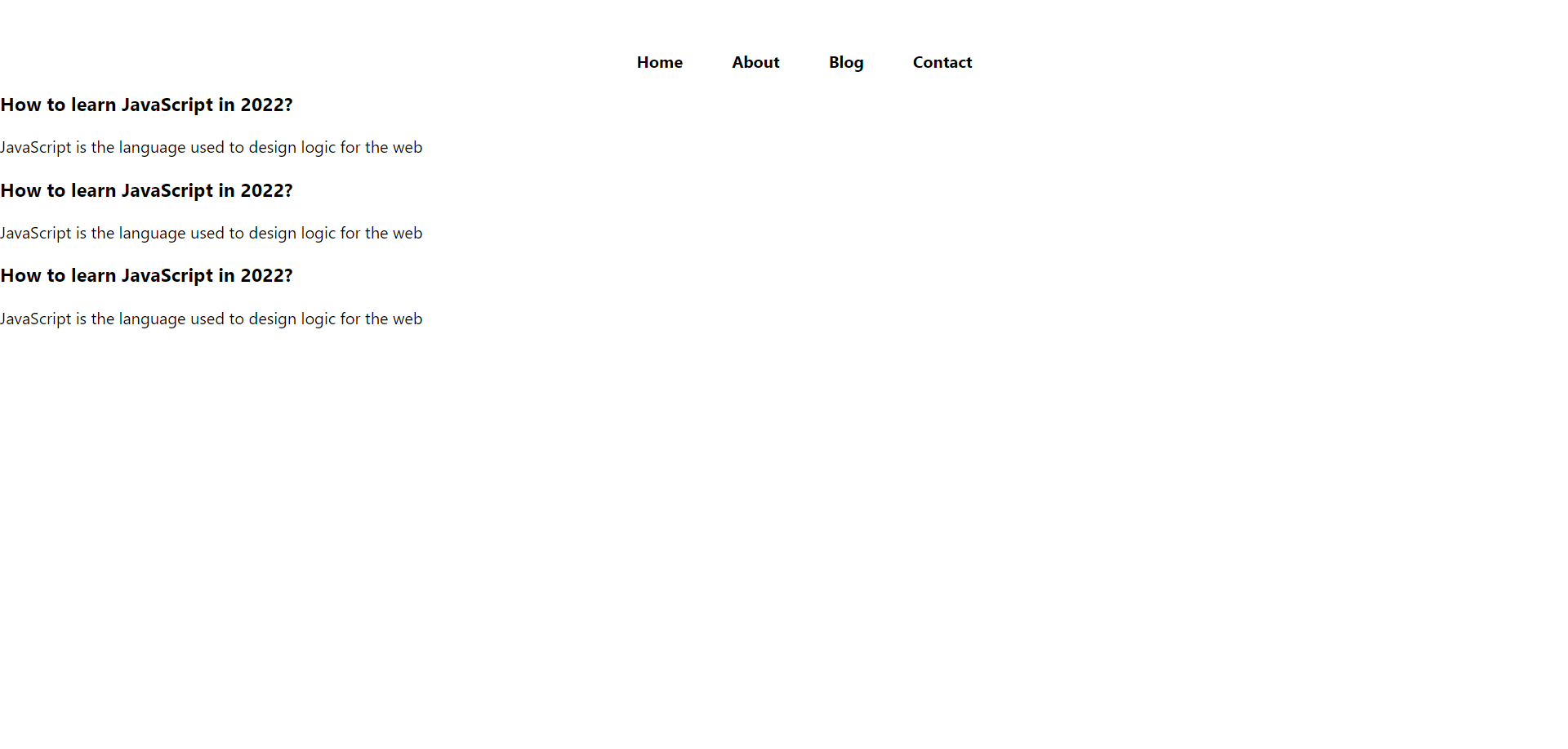
</div>;

};

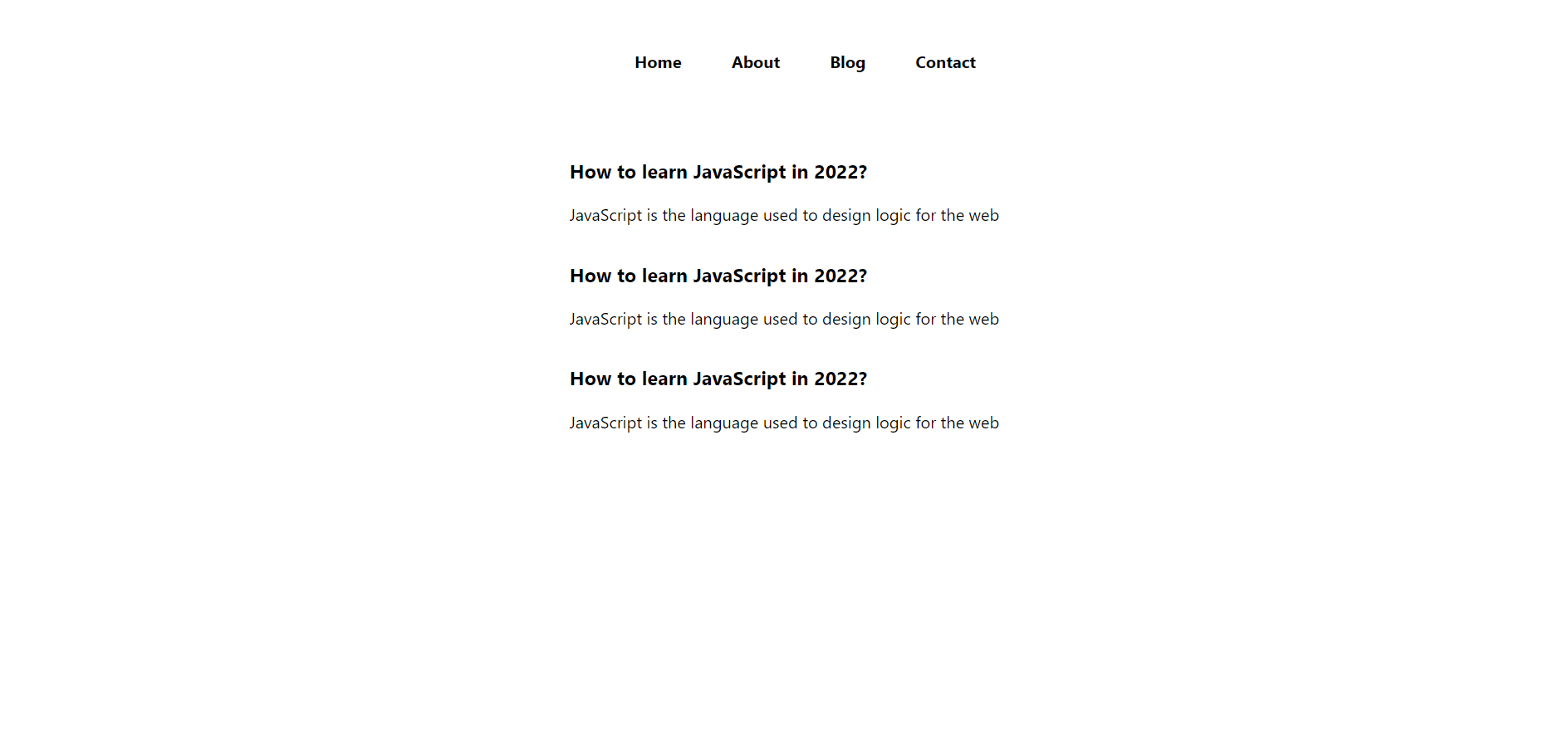
export default Blog;

**Code Snippet 1: Content of the blog.js file**

Although all the elements of the homepage make their way to the blog page after what we did above, the styling of this page falls far behind.



In order to accomplish that, we will create a separate style for the blog page. Get into the styles folder and create another file named Blog.module.css. The styling would be more or less similar to what we have on the homepage. Before we write the CSS here, import this .module.css file in the blog.js file. Since styling has never been our first priority, we’ll simply copy some of the classes from Home.module.css. Copy the main class and this should already put the whole content in the centre and with the same styling on the homepage. See the screenshot below.



We’ll now insert a Link component here to have the title for each of the blogs redirect to the blog page corresponding to it. To be able to use the Link component, make sure you have imported Link from next/link. Wrap the h3 tag inside this Link component and make it redirect to a blog post page exploiting the content of the slug. Add another class blogItemh3 in the Blog.module.css to make the cursor become a pointer when it hovers over the title of the blog. Here’s how we have changed these files.

.main {

min-height: 100vh;

flex: 1;

display: flex;

flex-direction: column;

justify-content: center;

align-items: center;

justify-content: flex-start;

padding-top: 3rem;

}

.blogItemh3{

cursor: pointer;

}

**Code Snippet 2: Blog.module.css**

import React from 'react';

import styles from '../styles/Blog.module.css'

import Link from 'next/link';

const Blog = () => {

return <div className={styles.container}>

<main className={styles.main}>

<div>

<Link href={'/blogpost/learn-javascript'}>

<h3 className={styles.blogItemh3}>How to learn JavaScript in 2022?</h3></Link>

<p>JavaScript is the language used to design logic for the web</p>

</div>

<div className="blogItem">

<h3>How to learn JavaScript in 2022?</h3>

<p>JavaScript is the language used to design logic for the web</p>

</div>

<div className="blogItem">

<h3>How to learn JavaScript in 2022?</h3>

<p>JavaScript is the language used to design logic for the web</p>

</div>

</main>

</div>

};

export default Blog;

**Code Snippet 3: blog.js**

This is where we finish creating our blog page. Currently, when you click the title of the first blog, you reach a page which is supposedly the blog post page for that blog. We, now, have to structure this one.

### Creating the Blog Page

Navigate to your blogpost folder, where the dynamic files assigned to each blog post are located. Open the [slug].js file. Clear everything written in the return statement of the slug function. Create an h1 tag for the title of the blog, followed by an hr tag for a line separating the title and the content. A div element containing the content of the blog would appear below the title.

To style this structure, we will create a separate style for the blog post page. Get into the styles folder and create another file named BlogPost.module.css.  Copy the main class from the Blog.module.css and paste it here. Wrap everything in the return statement in a div element with class main. We would also add a container class for additional padding. This is how both the CSS and JS files look.

import React from 'react';

import { useRouter } from 'next/router'

import styles from '../../styles/BlogPost.module.css';

const slug = () => {

const router = useRouter();

const { slug } = router.query;

return <div className={styles.container}>

<main className={styles.main}>

<h1>Title of the page {slug}</h1>

<hr />

<div>

Lorem ipsum dolor sit amet consectetur, adipisicing elit. Dolorem nulla repudiandae sint facilis, sunt corrupti numquam id illo. Ut deserunt animi iste voluptatum!

Lorem ipsum dolor sit amet consectetur adipisicing elit. Repellendus maxime rem earum repudiandae, cum possimus quae assumenda nulla culpa. Odit architecto repellendus non, unde recusandae placeat nisi perferendis quod nesciunt! Dolorum sapiente et sint consequuntur earum blanditiis iusto reprehenderit molestiae quia eligendi? Exercitationem, officia nobis!

</div>

</main>

</div>;

};

**Code Snippet 4: [slug].js in the blogpost folder**

.main {

min-height: 100vh;

flex: 1;

display: flex;

flex-direction: column;

justify-content: center;

align-items: center;

justify-content: flex-start;

padding-top: 3rem;

}

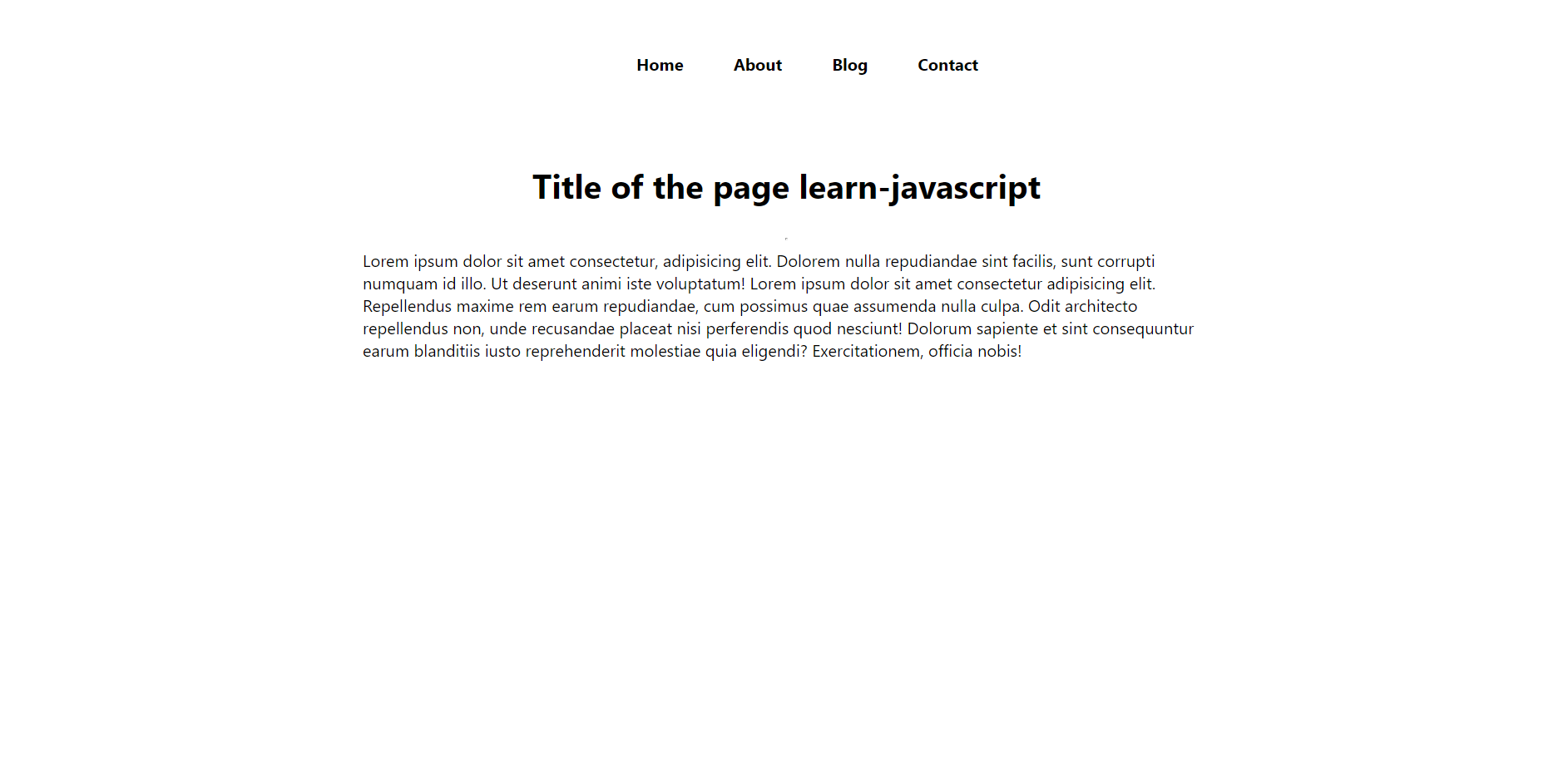
.container {

padding: 0 22rem;

}

**Code Snippet 5: BlogPost.module.css**

And this is where we finish the structuring of the blogpost page as well. This is how it looks.



# **Adding blog jsons as data**

These are steps we will follow to display title and description on the blog page.

1. First step would be to **collect all the files from the *blogdata* directory**.
2. Second step would be to**iterate through all the collected files and display the needed information on the *blog.js*page**.

These are steps we will follow to display the content of the blog on the blogpost page.

1. First step would be to **find the file in the *blogdata*directory corresponding to the slug**.
2. Second step would be to**populate the content of the blog inside the page in the format specified.**

# Introduction to API Routes in Next.js

You may recall we discussed how Next Js is not just a front-end framework, but a backend framework as well. Let's see how we can create API routes using Next Js to prove this claim right. We will see how we can exploit the api folder in the pages folder to construct our backend for the website. As we build the backend, we will also see how API endpoints can be utilised and how request and response objects can be handled. So, let’s just get started with the introduction of APIs.

As of now, the data files have been created. Let’s see how we can serve them. Run the development server. Now, we want to have all the JSON files present in the blogdata folder rendered and displayed on the blog page as a list and on the blogpost page when the corresponding slug is found in the query. For that, we will first have to get ourselves acquainted with the Next Js API.

There is a folder named api in the pages folder. The rest of the files in the pages folder make up the website's front-end. We will now be looking at the website's backend which is not visible to the user. In the api folder, there is a JavaScript file named hello. Now, when you search <http://localhost:3000/api/hello> on your browser, you get to see {"name":"John Doe"} which is exactly what has been given by the program hello.js below.

export default function handler(req, res) {

res.status(200).json({ name: 'John Doe' })

}

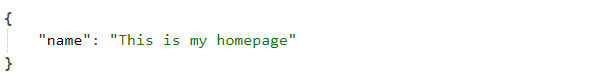
**Code Snippet 1: pages/api/hello.js**

This is how Next JS is already a backend framework now. We can build our desired API routes utilising the file-based routing principle. In this way, Next JS serves as both a front-end and a backend framework. Changes in the returned JSON statement in this file would also get reflected on the above link. Suppose, I replace John Doe with Harry Bhai. Below is what it shows now.



So, we just want to get the requested slug via some API endpoints and then retrieve the corresponding blog content and return it via the JSON statement. Backend is something no user can access. Even if you try logging your data in the console, it won’t get displayed there. That's the level of integrity almost every backend tool maintains. Although, things you log get printed in your private server, here, on your VSCode terminal. You can do that for both request(req) and response(res) objects.

As with how all pages get displayed according to the hierarchy they appear in the pages folder, which we call file-based routing, API routing follows the same principle. For example, create another file named index.js in the api folder. Copy the contents of the hello.js file and paste them here. Replace Harry Bhai with this is my homepage. And now, you can find this content on <http://localhost:3000/api>.



We can now implement all kinds of API we wish to in the same way. Be it a get request, post request or a put request. We will use them all according to our needs in the project.

# **Creating endpoints to get one/all blogposts**

Here, we will see how we can populate Hunting Coder with blogs and how a blog can be accessed using its slug. In addition, we'll see how accomplishing the above-mentioned tasks allows the website's frontend and backend to work together seamlessly. Basically, we wish to expose the information stored in the files in the blogdata folder via some endpoints that could be accessed by the file in the blogpost folder.

Having a basic knowledge of Node JS is not necessary for the job, but it would be nice if you did. First, rename the file hello.js that we built in the last tutorial to blogs.js. Access the file. Here, we wish to read the file system. For that, we will first clear whatever is already written inside the function. Exploring the actual documentation on reading a file in Next JS, we could find several ways to do that. One way is to use the fs module directly to interact with the file system. Import everything using a \* from ‘fs’ using the command below.

import \* as fs from 'fs';

Now, we could read individual files using the readFile function of fs. This function takes the path to the file as its first argument, the type of encoding system, UTF-8, as the second, and a callback function as the third. For demonstrating, we would use the how-to-learn-flask.json. file to get displayed. The callback function would first parse the data read by the readFile function into a string. Below is the snippet of the changes we made to the blogs.js file.

import \* as fs from 'fs';

export default function handler(req, res) {

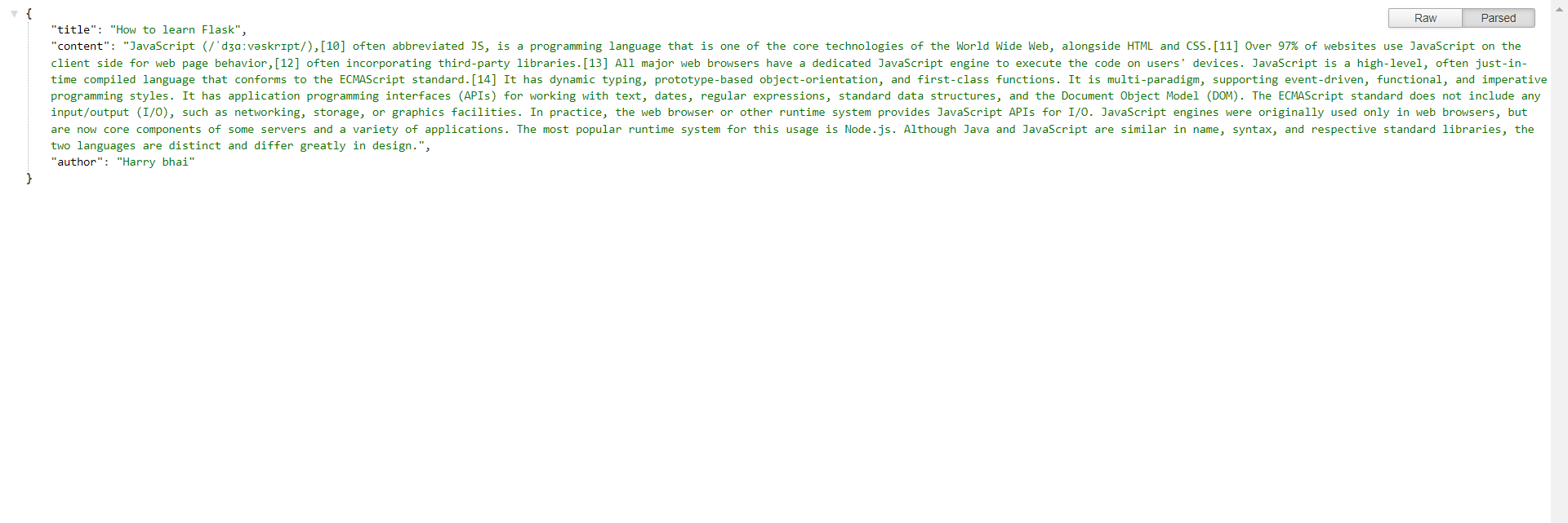
fs.readFile("blogdata/how-to-learn-flask.json", 'utf-8', (err, data) => {

res.status(200).json(JSON.parse(data));

})

}

And now, when we reload the page at <http://localhost:3000/api/blogs>, we would get to see the blog data corresponding to the file we used for reading.



But, if you could notice carefully, this is not what we originally wanted. We wanted to list the blog information corresponding to each of the files present in the blogdata directory and not only the ones we hardcoded in the backend, as here, only the first one got listed. Although, this function would also come into use when fetching information about individual blogs.

So, let’s just create another JS file, getblog.js, inside the same api folder and copy the method we used above in the blogs.js file to retrieve information about a single blog from JSON files present in the blogdata folder.

The function of this getblog API would be to feed the slug received from the query in the readFile function and search for a JSON file corresponding to the same slug in the blogdata folder and return its content in a JSON object format. Currently, the location of the file is hardcoded for the first JSON file. We will replace the name of the file with a variable that is nothing but the name of the file we’ll extract from the request query. We’ll also include an if statement, which will throw errors for invalid file names. Here’s the program for the getblog file.

import \* as fs from 'fs';

export default function handler(req, res) {

fs.readFile(`blogdata/${req.query.slug}.json`, 'utf-8', (err, data) => {

if (err) {

res.status(500).json({ error: "No such blog found" })

}

console.log(req.query.slug)

res.status(200).json(JSON.parse(data))

})

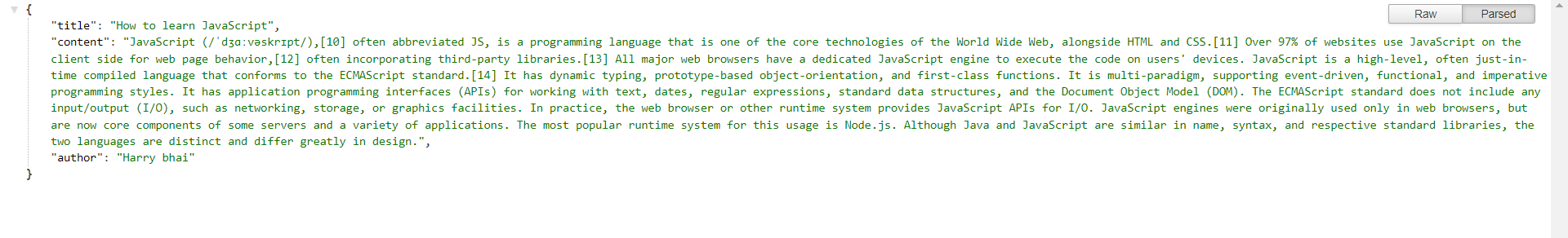
}

**Code Snippet 1:  pages/api/getblog.js**

So, now when we navigate to the page at <http://localhost:3000/api/getblog?slug=harry>, we get an error saying, no such blog found, since the slug harry is a filename that has no match in the blogdata folder.



But, if you look for the page at <http://localhost:3000/api/getblog?slug=how-to-learn-javascript>, we receive the JSON object corresponding to the file how-to-learn-javascript.json.



Let’s move back to blogs.js and make it work for listing all blogs present in the blogdata folder. For this, instead of using the function readFile, we’ll make use of the function readdir. This function takes the path to the directory as its first parameter and a callback function as the second. Here’s how the blogs.js API looks now.

import \* as fs from 'fs';

export default function handler(req, res) {

fs.readdir("blogdata", (err, data) => {

console.log(data)

res.status(200).json(data)

})

}

**Code Snippet 2:  pages/api/blogs.js**

And now, when you surf the page at <http://localhost:3000/api/blogs>, you’ll get to see the list of all the blogs present in the blogdata folder.



Here, we finish creating all our necessary API endpoints to retrieve different information for different purposes using the backend.

# **Using the API Endpoints**

Here we’ll see how we can use these endpoints in our frontend and establish a connection between the frontend and the backend. We’ll strategize the calling of these API routes and much more. So, let’s just get started.

Start by launching the development server. As of right now, you'll see a list of three blogs that are hard-coded into the program when you navigate to the blog page. And when you click on the first blog’s heading, it navigates you to a dummy page that has some garbage information that we hard-coded earlier. We wish to use the blogs API here to populate the actual list of blogs present on our server and the getblogs API to fetch the actual content of the corresponding blog.

### Recreating the blogs API for reading contents inside each blog

Currently, we only get to see the list of blogs present in the blogdata folder when we load the page at <http://localhost:3000/api/blogs>. But, that is not the only thing we want to read from the directory. We wish to read further the content of each of these blogs. For this, we will run a for loop iterating through all these blogs with a variable name item for each blog. Now, we’ll use the readFile function to read the content inside each of these blogs. This readFile function takes the path to the file as its first parameter and an encoding method as the second. Since we do not want to put the response here, but at the end, after reading all files, there would be no callback function. To store the content at this instant, we’ll need a variable container. So, define a container object named allBlogs at the beginning. Now, we can push the blog content into this array allBlogs and output the response containing the array once the loop has ended.

Here’s how the function looks now.

export default function handler(req, res) {

fs.readdir("blogdata", (err, data) => {

console.log(data)

let allBlogs = [];

data.forEach((item) => {

console.log(item)

fs.readFile(('blogdata/' + item), (d) => {

allBlogs.push(d)

})

})

})

res.status(200).json(allBlogs)

}

**Code Snippet 1: blogs API without the use of promises**

It is possible that most of us overlooked the fact that we didn't give enough time for the program to output desired results. The array might remain empty even after the program is compiled and run successfully. We’ll use the await-async method to prevent that error. We’ll also use promises to handle the delay caused by the program to read the files.

We’ll start by making this function asynchronous. Then, store the output given by the promised readdir function in a variable named data. Create another variable named myFile which will be used to hold the content of each blog. Run a JavaScript for loop that keeps running until the whole folder is read. Within the loop, define a variable called item to hold the current file value. Define another promised readFile function and feed it the required parameters and store the value it returns in myFile. Push the parsed information from myFile in JSON format in the array allBlogs. Once the loop ends, put the response as allBlogs. Here’s how the above function needed to be changed.

export default async function handler(req, res) {

let data = await fs.promises.readdir("blogdata");

let myfile;

let allBlogs = [];

for (let index = 0; index < data.length; index++) {

const item = data[index];

console.log(item)

myfile = await fs.promises.readFile(('blogdata/' + item), 'utf-8')

allBlogs.push(JSON.parse(myfile))

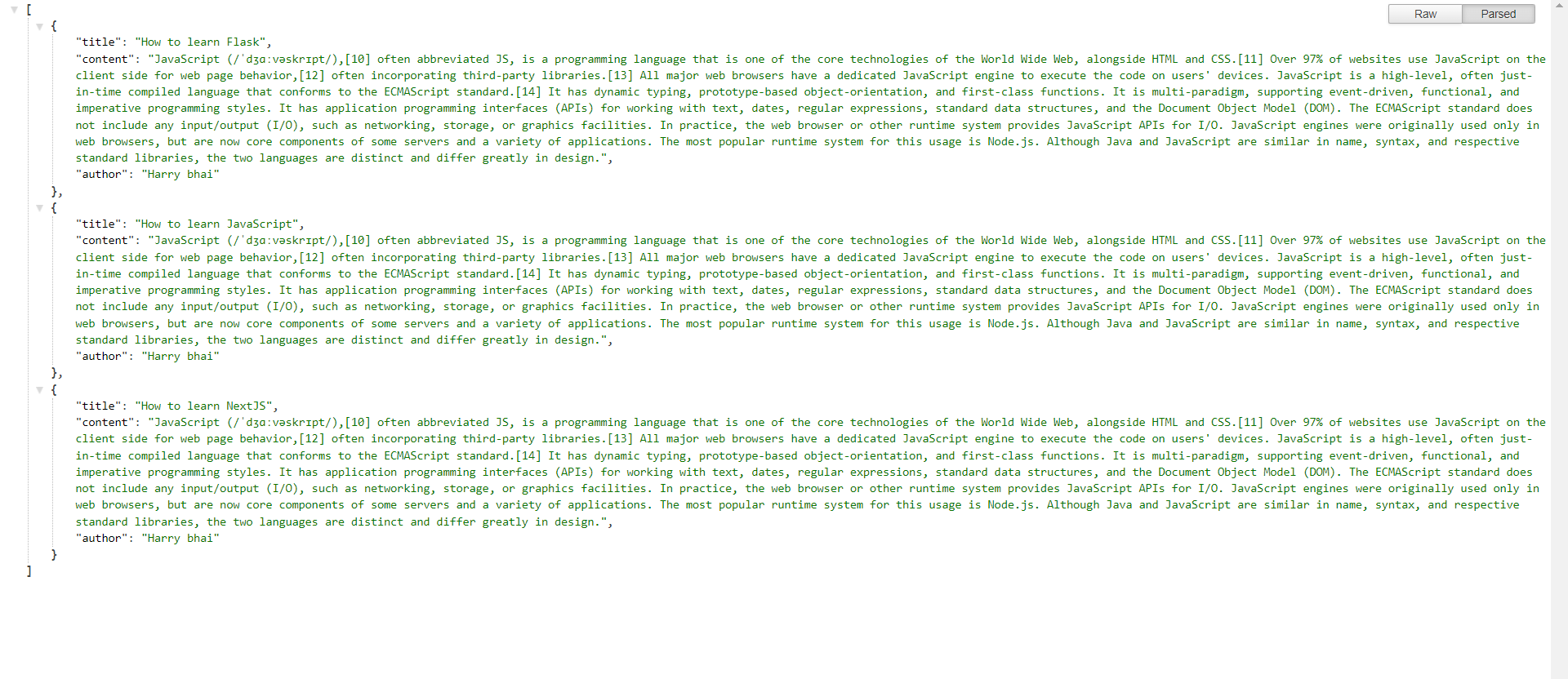
}

res.status(200).json(allBlogs)

}

**Code Snippet 2: blogs API with the use of promises**

And now, when you reload the page at  <http://localhost:3000/api/blogs>, you get to see the list of all blogs along with the content of those blogs.



### Using the blogs API

And now, we’ll make use of this blogs API in our front end to list the blogs on the blog page. Open the blog.js file present in the pages file. We can use useEffect Hook to fetch data from the API for our use in the frontend. Inside useEffect, use the fetch function to fetch the data from the blogs API. This fetch function returns a promise which further needs to be parsed and then returned. So, this function uses the .then method twice. Once for parsing and then for utilizing that parsed data. Define an empty state variable blogs using the useState Hook at the beginning. Set blogs to parsed.

Once this blogs state has all the parsed data, we’ll simply include this in the return statement of the Blog function using JavaScript map and the variable blogItem. We’ll add an element slug to all the JSON files to be able to use as a unique key for all blogs. Now, this blogItem could be easily used to return whatever JSX we were earlier returning, but this time, it would all be dynamic. We ensure that only an excerpt of the entire content gets displayed on the page. Here’s how the changed blog.js file looks.

import React, { useEffect, useState } from 'react';

import styles from '../styles/Blog.module.css'

import Link from 'next/link';

// Step 1: Collect all the files from the blogdata directory

// Step 2: Iterate through them and Display them

const Blog = () => {

const [blogs, setBlogs] = useState([]);

useEffect(() => {

console.log("useeffect is running");

fetch('http://localhost:3000/api/blogs').then((a) => {

return a.json();

})

.then((parsed) => {

console.log(parsed)

setBlogs(parsed)

})

}, [])

return <div className={styles.container}>

<main className={styles.main}>

{blogs.map((blogitem) => {

return <div key={blogitem.slug}>

<Link href={`/blogpost/${blogitem.slug}`}>

<h3 className={styles.blogItemh3}>{blogitem.title}</h3></Link>

<p className={styles.blogItemp}>{blogitem.content.substr(0, 140)}...</p>

</div>

})}

</main>

</div>

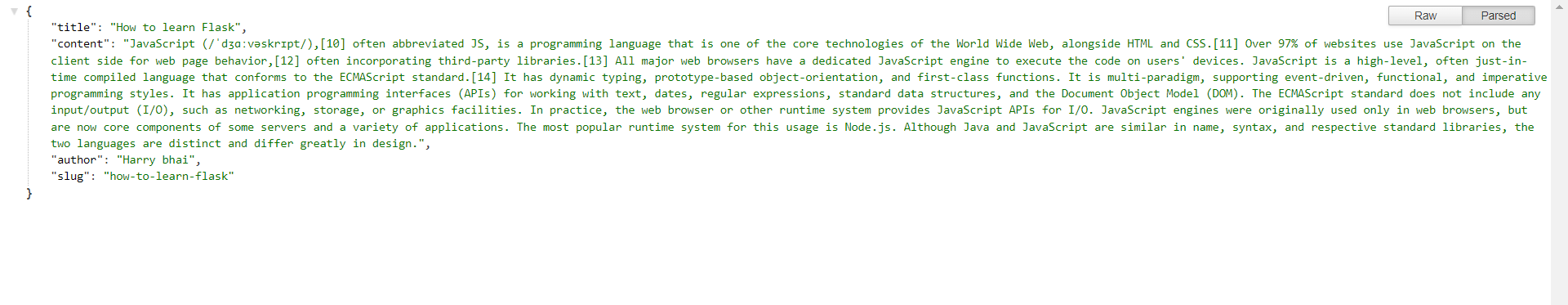
}

export default Blog;

# **Fetching BlogPost using /getblog Endpoint**

We will use another API named getblog to fetch the content corresponding to an individual blog when it is accessed from the website queried through a slug. The ideal way would be to use some database to do the same thing, but we will use the same file reading system for now. So, let’s just get started.

Run the development server.  At this point, when we click on a particular blog listed on the blog page, it navigates us to a page where the title is a slug and it is dynamic while the content is still hardcoded. So, currently, when you load the page at <http://localhost:3000/api/getblog?slug=how-to-learn-flask>, you get to see the content of the file how-to-learn-flask.json, and that is because the slug queried is valid and there is a file corresponding to that.



And the program throws an error stating that no such blog was found for slugs that are invalid. This is what we need to implement in the frontend using this API.

### Using the getblog API

Open the [slug].js file present in the blogpost folder. We will use the same useEffect method to fetch the information through the API. Define an empty state variable blog using the useState Hook before using the useEffect Hook. Inside useEffect, use the fetch function to fetch the data from the getblog API. Using the backticks and the slug, we'll give this function the location of the JSON file requested. The fetch function, again, returns a promise which further needs to be parsed and then returned. Set blog to parsed.

Once this blog state has all the parsed data, we’ll simply include this in the return statement of the slug function. We’ll replace the hardcoded title with blog.title and the hardcoded content with blog.content. However, this program would still throw errors since the fetch function would take enough time for this blog state to remain unset. To avoid that, we’ll render blog.title & blog.content only when the blog state is not empty, and we’ll also use the router.isReady boolean which ensures whether the router fields are updated client-side and ready for use. It was necessary to do this to prevent states from becoming empty due to an empty query. So, if (!router.isReady) return; would make sure that we don’t move ahead with the function until the router is ready to be used.

Here’s how the changed slug.js file looks.

import React, { useState, useEffect } from 'react';

import { useRouter } from 'next/router'

import styles from '../../styles/BlogPost.module.css';

// Step 1: Find the file corresponding to the slug

// Step 2: Populate them inside the page

const slug = () => {

const [blog, setBlog] = useState();

const router = useRouter();

useEffect(() => {

if (!router.isReady) return;

const { slug } = router.query;

fetch(`http://localhost:3000/api/getblog?slug=${slug}`).then((a) => {

return a.json();

})

.then((parsed) => {

setBlog(parsed)

})

}, [router.isReady])

return <div className={styles.container}>

<main className={styles.main}>

<h1>{blog && blog.title}</h1>

<hr />

<div>

{blog && blog.content}

</div>

</main>

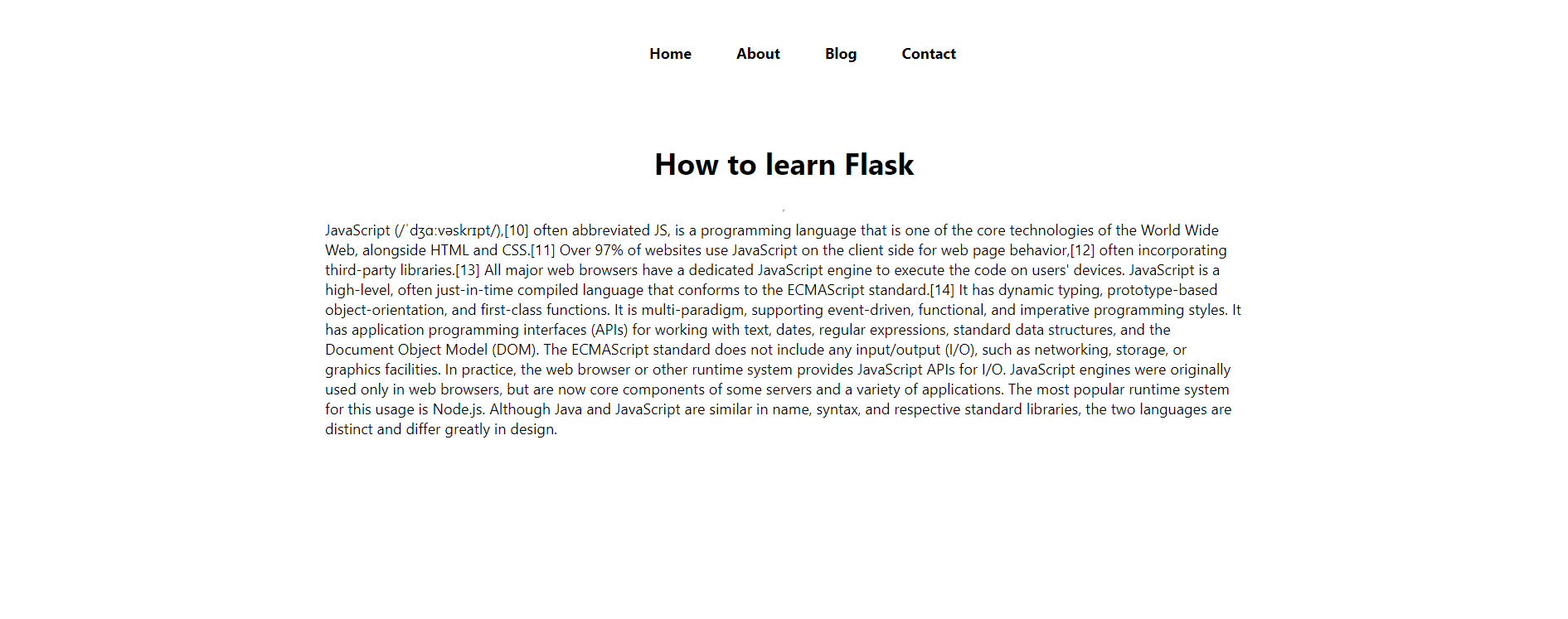
</div>;

};

export default slug;

**Code Snippet 1: blogpost/[slug].js**

And here’s how reloading the page at <http://localhost:3000/blogpost/how-to-learn-flask> looks. Contents are now dynamic, and things are looking pretty.



### Why is prerendering needed?

Suppose, we unanimously navigate to a blog page. Now, we could read the content of the blog page directly as it is being displayed on the front end. But, if I were to enact a bot, I would have to get to the page source to know what this page is all about. A bot gets to know all about the website only by retrieving the contents fed in the HTML of the page.

So, if this content is populated on the site using some API or JavaScript program and that program is down for some reason, our HTML fails to cast that content on the page source, and so does the content get missed by the bot even. And that is why we need our data to be pre rendered.

We need to ensure that our content is present even in the HTML of the webpage, since bots which are sometimes useful for the SEO of the website do not wait for the execution of the APIs or some JavaScript program.

Currently, our Hunting Coder doesn't feature that. In that case, if you navigate to this link <http://localhost:3000/blogpost/how-to-learn-flask> and view its source code, you wouldn't get the word flask on searching, which this blog is all about.



So, how would our bot know if this article is about flask? Or if the page has JavaScript written somewhere in its article? We will use the two pre rendering strategies we have. Prerendering are of two forms in Next JS - **Server-Side Rendering** and **Static Site Generations.**

### Server-Side Rendering

Here, the HTML files get hydrated on the server when the clients request them. That is, an HTML page gets developed, all before displaying them to the client, on the server, which has all the contents hydrated to them, and then, the server sends all the data to the clients on their demands, that is, as and when required.

And because of this, every time a client visits a page, it wouldn't have to wait for the execution of the JavaScript or for some APIs to respond. Our server guarantees the client an HTML page loaded with the content they are requesting.

Next JS has all the tools available to do this job efficiently. Server-side rendering is ideal for websites where requests cannot be pre-rendered, and the content changes too frequently, such as social media sites or where the feed is being updated live such as live news or cricket scores.

### Static Site Generation

Here, the whole front-end bundle of the website is stored in the form of a static site. That is, the whole front end of the website gets converted into a static site and it wouldn't have any role of the server in its rendering then.  Any Content Delivery Network (CDN) which is used to deliver web pages very quickly can now be used to deliver your site's web pages as well. Static Site delivery is very fast. Although, once you have deployed your static site after it is generated, any change in the website wouldn't get reflected until the site is regenerated and re-deployed, for the files get frozen once generated. This is one major drawback of Static Site Generation. This strategy is often used for websites whose content doesn't change too often as that of a blog page.

# **Server Side Rendering - getServerSideProps() in Next.js**

**Using *getServerSideProps()*on blog page*:***

So, using this *getServerSideProps*function on a page ensures the pre-rendering of the page on each request with the content, here *props,*returned by the function. Components can directly access the data returned by this function. So, define the *export async function getServerSideProps* and do whatever we earlier did in the *useEffect* function. Fetch the data from the *blogs* API and format it into JSON and return this data as *props* referring to the snippet below.

export async function getServerSideProps(context) {

let data = await fetch('http://localhost:3000/api/blogs')

let allBlogs = await data.json()

return {

props: { allBlogs }, // will be passed to the page component as props

}

**Code Snippet 1: Using the *getServerSideProps*on blog page**

Now, this *allBlogs* has all the content of all the blogs present on the server. We just have to feed this into the states we created in the *Blog*component. So, the changed *blog.js* file looks something like this.

import React, { useEffect, useState } from 'react';

import styles from '../styles/Blog.module.css'

import Link from 'next/link';

// Step 1: Collect all the files from blogdata directory

// Step 2: Iterate through the and Display them

const Blog = () => {

const [blogs, setBlogs] = useState([]);

useEffect(() => {

console.log("useeffect is running");

fetch('http://localhost:3000/api/blogs').then((a) => {

return a.json();

})

.then((parsed) => {

setBlogs(parsed)

})

}, [])

return <div className={styles.container}>

<main className={styles.main}>

{blogs.map((blogitem) => {

return <div key={blogitem.slug}>

<Link href={`/blogpost/${blogitem.slug}`}>

<h3 className={styles.blogItemh3}>{blogitem.title}</h3></Link>

<p className={styles.blogItemp}>{blogitem.content.substr(0, 140)}...</p>

</div>

})}

</main>

</div>

}

export async function getServerSideProps(context) {

let data = await fetch('http://localhost:3000/api/blogs')

let allBlogs = await data.json()

return {

props: { allBlogs }, // will be passed to the page component as props

}

}

export default Blog;

**Code Snippet 2: *pages/blog.js*after applying *getServerSideProps.***

Now, when you look at the page source of the same blog page, you would find the whole content there as well.



**Image 2: Page source after applying Server-Side Rendering**

Now, we’ll do the same thing for our blog post page as well.

**Using *getServerSideProps()*on blog page*:***

Open the *[slug].js* file present in the *blogpost*folder. Define the same *export async function getServerSideProps* and copy whatever we were earlier doing to fetch the content of a blog using its queried slug in the *useEffect* function. We have used the *context*parameter to receiver the *slug*here. Format the data into JSON and return this data as *props* referring to the snippet below.

export async function getServerSideProps(context) {

// console.log(context.query)

// const router = useRouter();

const { slug } = context.query;

let data = await fetch(`http://localhost:3000/api/getblog?slug=${slug}`)

let myBlog = await data.json()

return {

props: { myBlog }, // will be passed to the page component as props

}

}

**Code Snippet 2: Using the *getServerSideProps*on blog post page**

Similar to what we did on the blog page, we’ll delete whatever we were earlier doing on the client side. We will just feed this *prop*object to the states and that should suffice. Here’s how the *slug.js*page looks now.

import React, { useState, useEffect } from 'react';

import { useRouter } from 'next/router'

import styles from '../../styles/BlogPost.module.css';

// Step 1: Find the file corresponding to the slug

// Step 2: Populate them inside the page

const slug = (props) => {

const [blog, setBlog] = useState(props.myBlog);

return <div className={styles.container}>

<main className={styles.main}>

<h1>{blog && blog.title}</h1>

<hr />

<div>

{blog && blog.content}

</div>

</main>

</div>;

};

export async function getServerSideProps(context) {

// console.log(context.query)

// const router = useRouter();

const { slug } = context.query;

let data = await fetch(`http://localhost:3000/api/getblog?slug=${slug}`)

let myBlog = await data.json()

return {

props: { myBlog }, // will be passed to the page component as props

}

}

export default slug;

**Code Snippet 3: *pages/blogpost/[slug].js*after applying *getServerSideProps.***

**What are *getStaticProps & getStaticPaths?***

Here, we use two functions namely, *getStaticProps* and *getStaticPaths.*Since we are generating a static site of the whole website, we would need static pages for all the accessible pages present on the website, even the dynamic ones. So, *getStaticProps*returns a *props*object containing all the information it fetched from the database. Since APIs aren't available at the time the static site is generated, that part gets replaced by logic. Occasionally, a website will have dynamic pages, such as our blog post page. So, we need to define a list of paths to be statically generated for the *getStaticProps*function to make use of. And for that only, we use the *getStaticPaths*function.Once they are implemented in *Hunting Coder*, it will be clearer.

export async function getStaticProps(context) {

return {

props: {}, // will be passed to the page component as props

}

}

**Code Snippet 1: Syntax of a *getStaticProps*function**

export async function getStaticPaths() {

return {

paths: [

{ params: { ... } }

],

fallback: true // false or 'blocking'

};

}

**Code Snippet 2: Syntax of a *getStaticPaths*function**

**Using *getStaticPaths*in *[slug].js:***

Using *getStaticPaths*is very easy. We just have to list all the dynamic routes as a value to the *params*key in the *path* object of the return statement of the function. So, simply add the below snippet in the *[slug].js*file.

export async function getStaticPaths() {

return {

paths: [

{ params: { slug: 'how-to-learn-flask' } },

{ params: { slug: 'how-to-learn-javascript' } },

{ params: { slug: 'how-to-learn-nextjs' } },

],

fallback: true // false or 'blocking'

};

}

**Code Snippet 3: Using *getStaticPaths*in *pages/blogpost/[slug].js***

**Using *getStaticProps*in *[slug].js:***

Having listed the dynamic routes, we can feed the paths in the *getStaticProps*function. This function can access the paths via *context.params.*Earlier we were using the *getblog*API to fetch the blog data using the queried slug, but as I said earlier, API aren’t available there when we generate the static copy of the site. Consequently, we must substitute logical statements for the statements we used earlier. And for that, we will just have to copy the raw statements we wrote in the API files of the corresponding pages, here, the *getblog.js*. Here’s how the function looks.

export async function getStaticProps(context) {

const { slug } = context.params;

let myBlog = await fs.promises.readFile(`blogdata/${slug}.json`, 'utf-8')

return {

props: { myBlog: JSON.parse(myBlog) }, // will be passed to the page component as props

}

}

**Code Snippet 4: Using *getStaticProps*in *pages/blogpost/[slug].js***

**Using *getStaticProps*in *blog.js:***

Similar to what we did above, we’ll simply replace the content of the function with that of the logic of the *blogs*API. Here’s how it looks.

export async function getStaticProps(context) {

let data = await fs.promises.readdir("blogdata");

let myfile;

let allBlogs = [];

for (let index = 0; index < data.length; index++) {

const item = data[index];

console.log(item)

myfile = await fs.promises.readFile(('blogdata/' + item), 'utf-8')

allBlogs.push(JSON.parse(myfile))

}

return {

props: { allBlogs }, // will be passed to the page component as props

}

}

**Code Snippet 5: Using *getStaticProps*in *pages/blog.js***

Please adhere to the changes we have made while replacing the APIs with logical statements. Don’t forget to include the file read system module *fs.*

**Additional changes**

1. You cannot use the Image component of NextJS while building a static site. So, we’ll replace that with the *<img>*Here’s the changed image component.

{/\* <Image className={styles.myImg} src="/homeimg.jfif" width={237} height={158}/> \*/}

<img className={styles.myImg} src="/homeimg.jfif" width={237} height={158} alt="hunting coder" />

**Code Snippet 6: Replacing the Image component**

2. Now, we have written the program for generating the static site of the whole website, but we still need a script to enable the export feature of the website. So, open *json*and add the following code in the *scripts*object.

"export": "next build && next export"

3. While our website on running the command *yarn export*would generate a static copy of itself and store it in an *out*directory, the files there, on opening, would strictly want you to put the *.html* extension while browsing every time. To avoid that, we will put this code in the *exports*object of the *next.config.js*file.

trailingSlash: true

Here’s the changed version of the *blog.js*and *[slug].js*files.

import React, { useEffect, useState } from 'react';

import styles from '../styles/Blog.module.css'

import Link from 'next/link';

import \* as fs from 'fs';

// Step 1: Collect all the files from blogdata directory

// Step 2: Iterate through the and Display them

const Blog = (props) => {

console.log(props)

const [blogs, setBlogs] = useState(props.allBlogs);

// useEffect(() => {

// }, [])

return <div className={styles.container}>

<main className={styles.main}>

{blogs.map((blogitem) => {

return <div key={blogitem.slug}>

<Link href={`/blogpost/${blogitem.slug}`}>

<h3 className={styles.blogItemh3}>{blogitem.title}</h3></Link>

<p className={styles.blogItemp}>{blogitem.content.substr(0, 140)}...</p>

</div>

})}

</main>

</div>

};

export async function getStaticProps(context) {

let data = await fs.promises.readdir("blogdata");

let myfile;

let allBlogs = [];

for (let index = 0; index < data.length; index++) {

const item = data[index];

console.log(item)

myfile = await fs.promises.readFile(('blogdata/' + item), 'utf-8')

allBlogs.push(JSON.parse(myfile))

}

return {

props: { allBlogs }, // will be passed to the page component as props

}

}

export default Blog;

**Code Snippet 7: *pages/blog.js***

import React, { useState, useEffect } from 'react';

import { useRouter } from 'next/router'

import styles from '../../styles/BlogPost.module.css'

import \* as fs from 'fs';

// Step 1: Find the file corresponding to the slug

// Step 2: Populate them inside the page

const Slug = (props) => {

const [blog, setBlog] = useState(props.myBlog);

return <div className={styles.container}>

<main className={styles.main}>

<h1>{blog && blog.title}</h1>

<hr />

<div>

{blog && blog.content}

</div>

</main>

</div>;

};

export async function getStaticPaths() {

return {

paths: [

{ params: { slug: 'how-to-learn-flask' } },

{ params: { slug: 'how-to-learn-javascript' } },

{ params: { slug: 'how-to-learn-nextjs' } },

],

fallback: true // false or 'blocking'

};

}

export async function getStaticProps(context) {

const { slug } = context.params;

let myBlog = await fs.promises.readFile(`blogdata/${slug}.json`, 'utf-8')

return {

props: { myBlog: JSON.parse(myBlog) }, // will be passed to the page component as props

}

}

export default Slug;

**Code Snippet 8: *pages/blogpost/[slug].js***

You can now open the out directory created by next as a CDN directory and open it in new instance of VSCode and run the index file via Live server extension to see our website’s Static generated format.

**Rendering HTML text using dangerouslySetInnerHTML in Next.js**

**What is *dangerouslySetInnerHTML?***

When we render simple HTML, the content of the website gets delivered along with the HTML tags which doesn't give that great experience. The *dangerouslySetInnerHTML*method helps us put HTML code inside a container object. We will use it to modify the content of our blogs and replace its plain paragraphs and texts with some HTML content for it will give us more formatting options. So, let’s just get started.

**Using dangerouslySetInnerHTML**

Currently, all our content on the blog page are simple texts or paragraphs. Therefore, even if you replace the content currently there with HTML, you wouldn't see what you expect. Suppose, we replace the simple text in the *content*of the *how-to-learn-flask.json*file with some random HTML code I picked from some HTMl editor and save it.

{

"title": "How to learn flask",

"content": "<!-- ####### YAY, I AM THE SOURCE EDITOR! #########--> <h1 style=\/\"color: #5e9ca0;\/\">You can edit <span style=\"color: #2b2301;\">this demo<\/span> text!<\/h1> <h2 style=\"color: #2e6c80;\">How to use the editor:<\/h2> <p>Paste your documents in the visual editor on the left or your HTML code in the source editor in the right. <br \/>Edit any of the two areas and see the other changing in real time.&nbsp;<\/p> <p>Click the <span style=\"background-color: #2b2301; color: #fff; display: inline-block; padding: 3px 10px; font-weight: bold; border-radius: 5px;\">Clean<\/span> button to clean your source code.<\/p> <h2 style=\"color: #2e6c80;\">Some useful features:<\/h2> <ol style=\"list-style: none; font-size: 14px; line-height: 32px; font-weight: bold;\"> <li style=\"clear: both;\"><img style=\"float: left;\" src=\"https:\/\/html-online.com\/img\/01-interactive-connection.png\" alt=\"interactive connection\" width=\"45\" \/> Interactive source editor<\/li> <li style=\"clear: both;\"><img style=\"float: left;\" src=\"https:\/\/html-online.com\/img\/02-html-clean.png\" alt=\"html cleaner\" width=\"45\" \/> HTML Cleaning<\/li> <li style=\"clear: both;\"><img style=\"float: left;\" src=\"https:\/\/html-online.com\/img\/03-docs-to-html.png\" alt=\"Word to html\" width=\"45\" \/> Word to HTML conversion<\/li> <li style=\"clear: both;\"><img style=\"float: left;\" src=\"https:\/\/html-online.com\/img\/04-replace.png\" alt=\"replace text\" width=\"45\" \/> Find and Replace<\/li> <li style=\"clear: both;\"><img style=\"float: left;\" src=\"https:\/\/html-online.com\/img\/05-gibberish.png\" alt=\"gibberish\" width=\"45\" \/> Lorem-Ipsum generator<\/li> <li style=\"clear: both;\"><img style=\"float: left;\" src=\"https:\/\/html-online.com\/img\/6-table-div-html.png\" alt=\"html table div\" width=\"45\" \/> Table to DIV conversion<\/li> <\/ol> <p>&nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp;<\/p> <h2 style=\"color: #2e6c80;\">Cleaning options:<\/h2> <table class=\"editorDemoTable\"> <thead> <tr> <td>Name of the feature<\/td> <td>Example<\/td> <td>Default<\/td> <\/tr> <\/thead> <tbody> <tr> <td>Remove tag attributes<\/td> <td><img style=\"margin: 1px 15px;\" src=\"images\/smiley.png\" alt=\"laughing\" width=\"40\" height=\"16\" \/> (except <strong>img<\/strong>-<em>src<\/em> and <strong>a<\/strong>-<em>href<\/em>)<\/td> <td>&nbsp;<\/td> <\/tr> <tr> <td>Remove inline styles<\/td> <td><span style=\"color: green; font-size: 13px;\">You <strong style=\"color: blue; text-decoration: underline;\">should never<\/strong>&nbsp;use inline styles!<\/span><\/td> <td><strong style=\"font-size: 17px; color: #2b2301;\">x<\/strong><\/td> <\/tr> <tr> <td>Remove classes and IDs<\/td> <td><span id=\"demoId\">Use classes to <strong class=\"demoClass\">style everything<\/strong>.<\/span><\/td> <td><strong style=\"font-size: 17px; color: #2b2301;\">x<\/strong><\/td> <\/tr> <tr> <td>Remove all tags<\/td> <td>This leaves <strong style=\"color: blue;\">only the plain<\/strong> <em>text<\/em>. <img style=\"margin: 1px;\" src=\"images\/smiley.png\" alt=\"laughing\" width=\"16\" height=\"16\" \/><\/td> <td>&nbsp;<\/td> <\/tr> <tr> <td>Remove successive &amp;nbsp;s<\/td> <td>Never use non-breaking spaces&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;to set margins.<\/td> <td><strong style=\"font-size: 17px; color: #2b2301;\">x<\/strong><\/td> <\/tr> <tr> <td>Remove empty tags<\/td> <td>Empty tags should go!<\/td> <td>&nbsp;<\/td> <\/tr> <tr> <td>Remove tags with one &amp;nbsp;<\/td> <td>This makes&nbsp;no sense!<\/td> <td><strong style=\"font-size: 17px; color: #2b2301;\">x<\/strong><\/td> <\/tr> <tr> <td>Remove span tags<\/td> <td>Span tags with <span style=\"color: green; font-size: 13px;\">all styles<\/span><\/td> <td><strong style=\"font-size: 17px; color: #2b2301;\">x<\/strong><\/td> <\/tr> <tr> <td>Remove images<\/td> <td>I am an image: <img src=\"images\/smiley.png\" alt=\"laughing\" \/><\/td> <td>&nbsp;<\/td> <\/tr> <tr> <td>Remove links<\/td> <td><a href=\"https:\/\/html-online.com\" rel=\"nofollow\">This is<\/a> a link.<\/td> <td>&nbsp;<\/td> <\/tr> <tr> <td>Remove tables<\/td> <td>Takes everything out of the table.<\/td> <td>&nbsp;<\/td> <\/tr> <tr> <td>Replace table tags with structured divs<\/td> <td>This text is inside a table.<\/td> <td>&nbsp;<\/td> <\/tr> <tr> <td>Remove comments<\/td> <td>This is only visible in the source editor <!-- HELLO! --><\/td> <td><strong style=\"font-size: 17px; color: #2b2301;\">x<\/strong><\/td> <\/tr> <tr> <td>Encode special characters<\/td> <td><span style=\"color: red; font-size: 17px;\">&hearts;<\/span> <strong style=\"font-size: 20px;\">\u263A \u2605<\/strong> &gt;&lt;<\/td> <td><strong style=\"font-size: 17px; color: #2b2301;\">x<\/strong><\/td> <\/tr> <tr> <td>Set new lines and text indents<\/td> <td>Organize the tags in a nice tree view.<\/td> <td>&nbsp;<\/td> <\/tr> <\/tbody> <\/table> <p><strong>&nbsp;<\/strong><\/p> <p><strong>Save this link into your bookmarks and share it with your friends. It is all FREE! <\/strong><br \/><strong>Enjoy!<\/strong><\/p> <p><strong>&nbsp;<\/strong><\/p>",

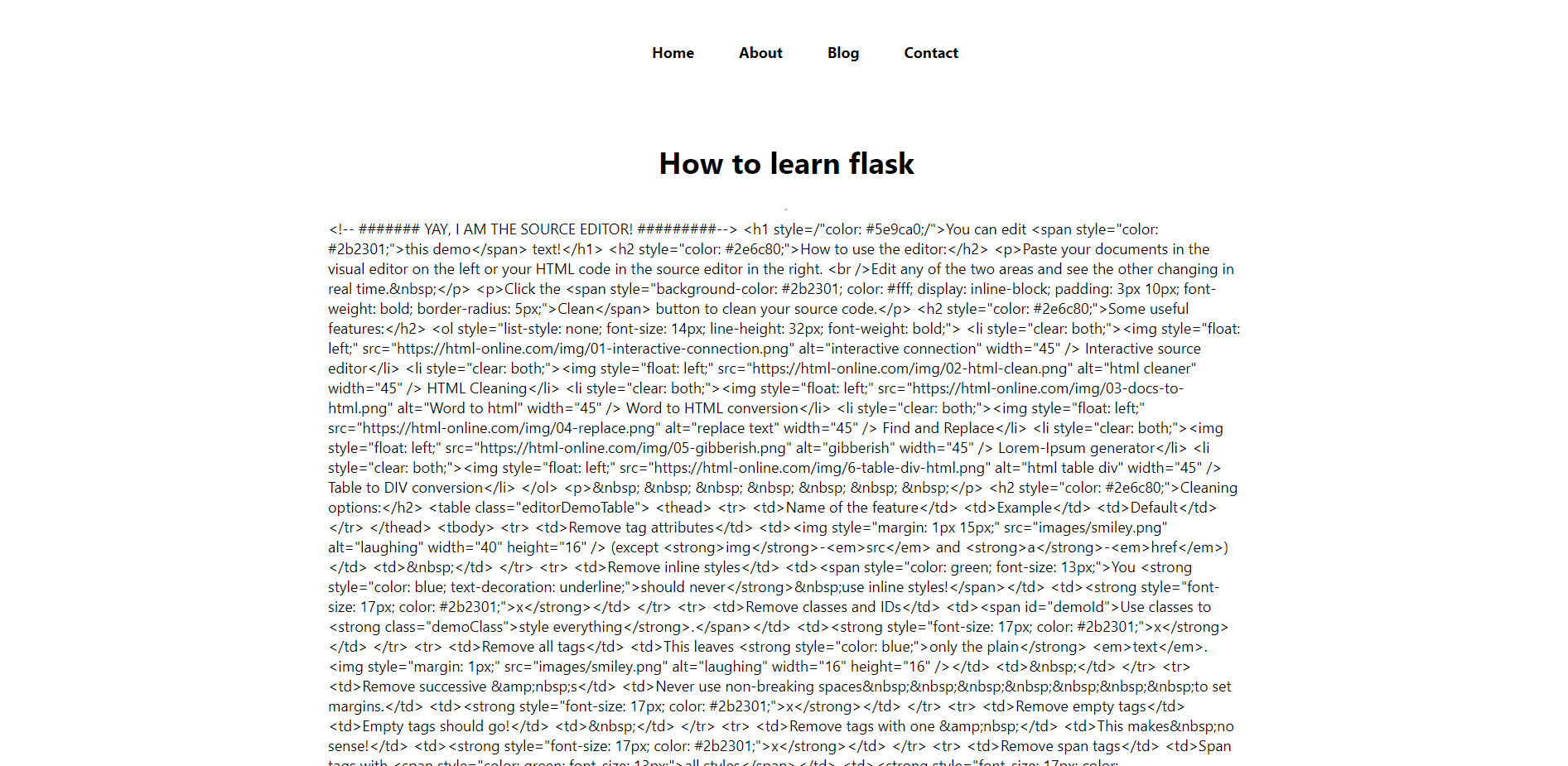
"author": "Harry bhai",

"slug": "how-to-learn-flask"

}

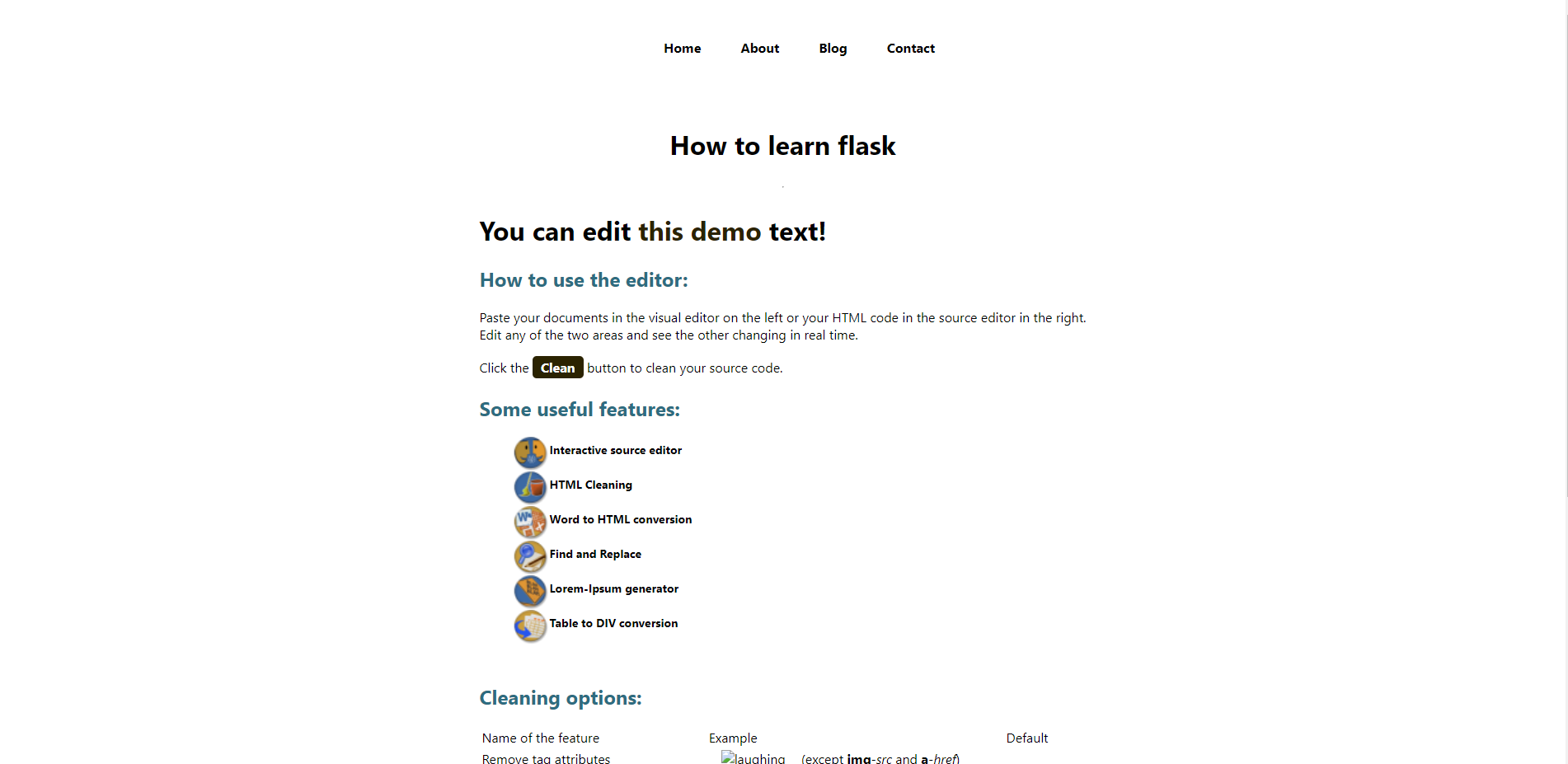
**Code Snippet 1: *how-to-learn-flask.json***

Here’s what you will get to see on the page displaying this content.



There was nothing more than a simple text display. We wish to replace the content of these pages with some HTML and at the same time, retain their format while rendering. We want to have this HTML content parsed and that is when we use the React tool named *dangerouslySetInnerHTML.*

See the HTML content rendered as expected.



Here’s how the changed *Slug*component looks.

const Slug = (props) => {

function createMarkup(c) {

return { \_\_html: c };

}

const [blog, setBlog] = useState(props.myBlog);

return <div className={styles.container}>

<main className={styles.main}>

<h1>{blog && blog.title}</h1>

<hr />

{blog && <div dangerouslySetInnerHTML={createMarkup(blog.content)}></div>}

</main>

</div>;

};

**Code Snippet 5: *Slug*component after applying *dangerouslySetInnerHTML***