**Documentation link:**

[**https://nuxt.com/docs/getting-started/introduction**](https://nuxt.com/docs/getting-started/introduction)

**Prerequisites**

• node - We recommend you have either 16.x or 14.x installed.

• A text editor, we recommend VS Code with the Volar extension.

• A terminal, we recommend using VS Code's integrated terminal.

*//https://tailwindcss.nuxtjs.org/getting-started/setup*

*//to add tailwind css to nuxt project*

**Command to create new project for nuxt 3**

**With NPM**

**npx create-nuxt-app@latest (installs nuxt 2)**

**With NPX**

**npx nuxi@latest init <project-name> (installs nuxt 3)**

then do **npm install**

then to open development server type **npm run dev -- -o**

**npm create vite@latest -> use this command to get a vue 3 project**

**UNDERSTANDING DIRECTORY STRUCTURE:**

1. Nuxt.config.ts: - this file should always be .ts file (type script file). This file is used to extend or change the default NUXT configuration for your project. You can add custom meta data here, register modules here or add api keys here.

**export default defineNuxtConfig({**

**devtools: { enabled: true },**

**modules: ['@nuxtjs/tailwindcss']**

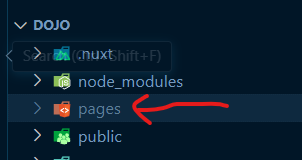
**})**

Like for example here modules: ['@nuxtjs/tailwindcss'] is used to add tailwind css. You can add more modules in modules array just make it comma separated

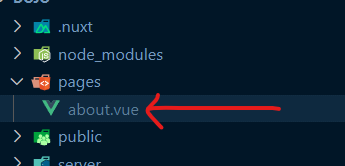
1. App.vue: - This is the default root component of the application. Means it is like index.php (entry point or starting point of an application). It’s the home page of application. You can see **<NuxtWelcome />** in this component which is displaying the default nuxt home page. To start making your app. Remove this component.
2. . nuxt folder: - the nuxt folder is used by nuxt joint development to generate the nuxt application. We do not need to change anything in this folder, leave it as it is.
3. . gitignore: - this is the file used to control what needs to be ignored while pushing to the remote repository.
4. Package. Json: - package. Json file is where you can see all the JavaScript libraries and dependencies installed and their version.

**Adding More pages to application:**

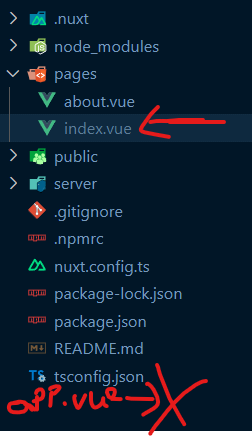
1. Create a folder named pages in root directory



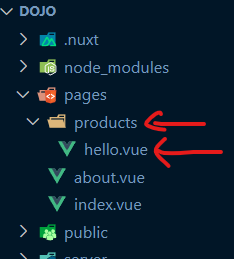
1. Inside this page’s folder we will create different .Vue files. Each file will represent different page and routes. And the route path will be determined by file name. For example, if we create a component inside the pages folder called about. Vue, then nuxt will automatically create a route for this page which will be /about in the URL. And if we visit this route localhost:3000/about we will see the about component displaying in the browser.



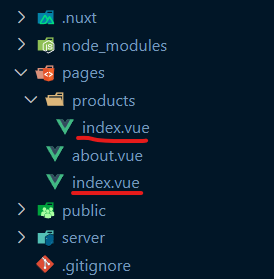
1. In order to let Nuxt use our page components in pages folder we must either delete the app.vue component or use a special component inside app.vue component called NuxtPage (like router-view in app.js in vue).
2. In our case we will delete the app.vue file (which behaves as a home page for now). And create a new file inside pages folder named index.vue file. this file (index.vue) will now behave as a home page for us now. Nuxt will no need to create a /index route for index.vue file, instead the route for index.vue file will be **“/”, home page and no need to use** NuxtPage



1. Now write some code in about.vue and index.vue files (vue code (type vbase and press enter)). And start the development server by typing npm run dev in terminal. You will notice that In Browser, you will see home page on this URL localhost:3000, and about page on localhost:3000/about URL
2. If we create subfolders in pages directory, Nuxt will automatically add the name of the subfolder to the route path of the subfolder files. For example, if we create a Products folder inside pages folder, then if we add a component inside Products folder, then the name of the “Products” will be added to the route path for all pages inside the Products folder. For example, if we create a file named hello.vue inside pages/products folder. Then the path for this hello.vue file will be /products/hello. And when we visit localhost:3000/products/hello, the hello.vue file will be rendered.



1. In our scenario we do not need hello.vue file inside pages/products folder. So, we will delete hello.vue. And instead create a file inside pages/products folder named index.vue. now this file will generate the following route for us localhost:3000/products. Same like the home page for the website is index.vue file inside the pages folder, the index.vue file inside the products folder, is the home page for the sub folder or subdomain.

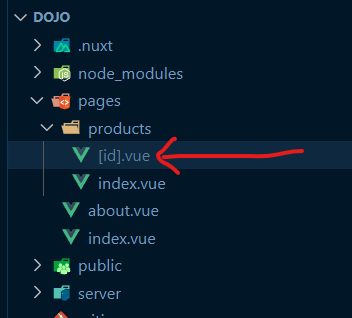


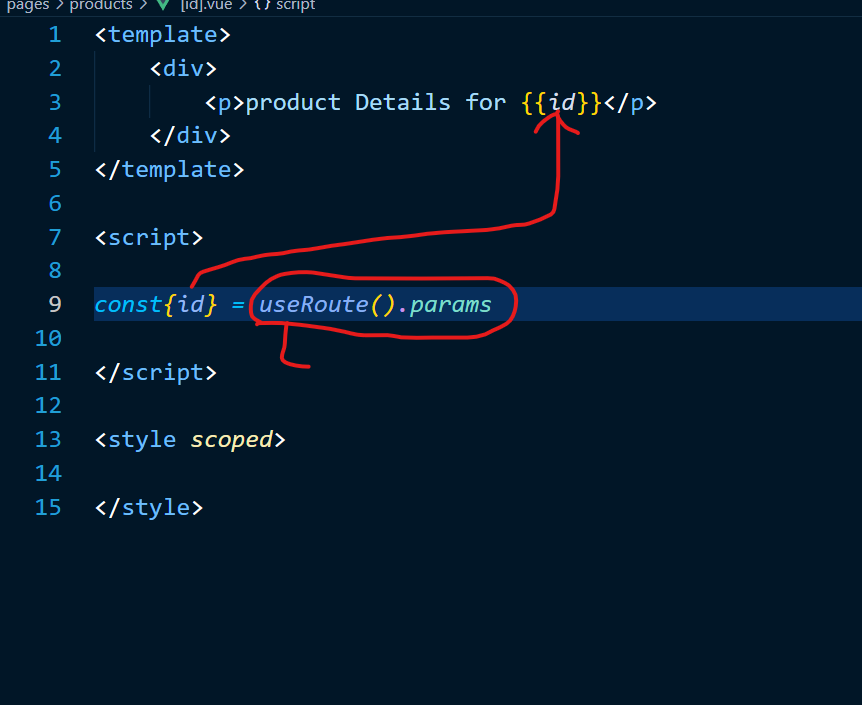
**Adding Dynamic Routes in Nuxt using ID Example:**

**localhost:3000/products/1, localhost:3000/products/2: -**

1. The routes declared in the previous section are definite or static routes, they are not dynamic. What if we want dynamic routing for URL like localhost:3000/products/id, where “id” is the product id (1,2,3,4 etc.), and “id” is dynamic and changeable. We want to use the same structure of HTML and CSS to display every product, no matter what ever the product id is. So, we will be using the same component structure (html, CSS) inside the component, but the data will change based on the ID we pass to the component. So, products page is where you can see all the products localhost:3000/products, but when you click on one product on product page, you will see a page for that product details page specifically localhost:3000/products/id.

1. To achieve this dynamic routing, we will create a file inside a sub folder, for which we want to display data for different Id’s. The name of this file will be wrapped inside “[]”. We can name whatever we want but we will be using id here. We name it [id]. Vue here



1. Now when we visit localhost:3000/products/id**,** the [id]. vue component will be rendered. Note here that “id” here in the URL can be anything, it can be 1,2,3 or a string like Mario or Hamza.
2. [id]. Vue file will pick up all the route variation of the id, because we used the square brackets “[ ]” as part of the file name. and this tells the Nuxt that this part of the route (route parameter) is changeable. So, it does not matter whatever the value of “id” is in the URL, we will still display the [id]. vue component
3. We can capture the value of “id” inside [id]. Vue file by using use route composable function.

Here useRoute().params is the use route composable function.

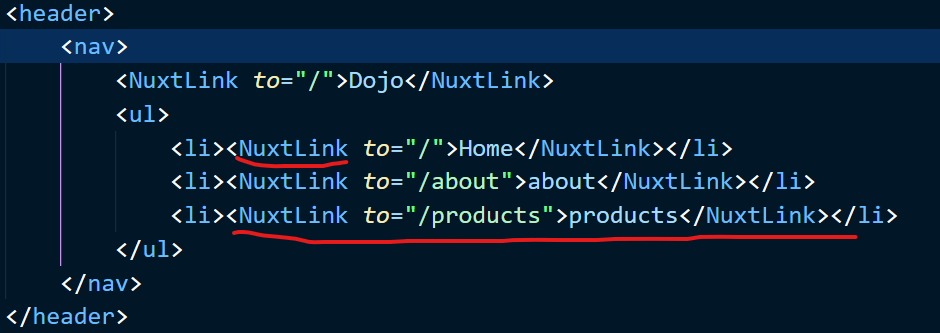
Note here that the name inside const {id} must match the name inside [id] file name.

In Vue3 and Nuxt 3 we cannot use $this.route to get access to the URL parameters, so we use Router composable, useRouter and useRoute .

* route, which will contain all the info related to the path, hostname, current query params etc...
* router, the actual Vue router that will provide you all the features such as a client side navigation (with push or replace), rollback to the n-1, resolve etc...

**Navbar in Nuxt using NuxtLink:**

We use <NuxtLink></NuxtLink> instead of anchor tag(<a></a>) for routing to another page



**Managing Layouts in Nuxt (header and footer):**

1. To manage layouts, create a folder in root directory called Layouts. You can have many types of layouts. But to create a default layout you need to create a file inside the layout folder named default.vue.

paste the following code in default.vue file.

<header *class*="shadow-sm bg-white">

            <nav *class*="container mx-auto p-4 flex justify-between">

                <NuxtLink *to*="/" *class*="font-bold">Dojo</NuxtLink>

                <ul *class*="flex gap-4">

                    <li><NuxtLink *to*="/">Home</NuxtLink></li>

                    <li><NuxtLink *to*="/about">about</NuxtLink></li>

                    <li><NuxtLink *to*="/products" *class*="btn">products</NuxtLink></li>

                </ul>

            </nav>

</header>

Now you want to make use of this layouts/default.vue file in different pages. How can we do that?

Let’s consider that layout/default.vue file is the child component of all pages. So, it means we can pass down the HTML content to display to it using <slot/>.

By Using vue </slot> component, the header code inside default.vue file will be displayed on all of the pages of nuxt, even on localhost:3000/products/123. <slot / > is a built-in component in Nuxt, so when Nuxt sees this <slot / > component inside a layout folder, it automatically knows that it needs to display HTML content coming from pages folder on an active page. Conceptually speaking layout folder is a child of pages folder.

## Slot

We have learned that components can accept props, which can be JavaScript values of any type. But how about HTML template content? In some cases, we may want to pass a template fragment to a child component, and let the child component render the fragment within its own template. Like props are for passing data to child component, slots are used to pass Html content to the child component. We write <slot /> in the child component as any data can be passed down to child component to be displayed. This means Layout/default.vue file is a child component of the files inside pages folder.

For detail listen to following lecture: <https://www.udemy.com/course/vuejs-2-the-complete-guide/learn/lecture/21526198#overview>

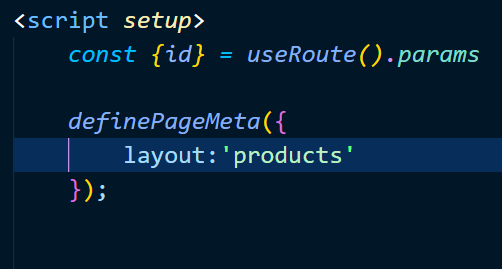
1. Currently we have a default layout for all the pages of the application. What if we want to create different layout for some pages?

To create custom layouts, create a new file in layouts folder for example products. Vue (we can name it anything we want names do not have to match with subfolder name), because we want a different layout for all the product pages.

Change the layout to your desire in layout/products.vue file and save it.

Now to apply this different layout to the different pages in pages folder. for example, if we want to apply layouts/products.vue layout to products/index.vue file, we can use a special composable function that Nuxt Gives us inside the <script setup></script> called

definePageMeta. this function accepts an object as a parameter, and there we can specify layout property and then mention the layout we want to use from the layout folder, in that page.



**To add tailwind CSS to Nuxt Project follow the link below**

[https://tailwindcss.nuxtjs.org/getting-started/setup#](https://tailwindcss.nuxtjs.org/getting-started/setup)

command to install tailwind CSS:

npm install --save-dev @nuxtjs/tailwindcss

then add the tailwind CSS module to Nuxt Config file.

Inside nuxt.config.ts file add the module by typing  
  
modules: [‘@nuxtjs/tailwindcss’]

inside

*export* *default* *defineNuxtConfig* ({

Now if you want to add more modules. You can comma separate it. And each value must be a string value.

modules: [

‘@nuxtjs/tailwindcss’,

‘asdasdasdasd’,

‘asdasdasd’

]

After this, create a folder in root directory named assets. Inside assets create a folder named CSS. Then create a file inside CSS folder named tailwind.css

Then open tailwind.css and paste the following inside the file

@tailwind base;

@tailwind components;

@tailwind utilities;

then create a tailwind config file by using tailwind init (optional).

**Fetching external data and using in components:**

We use useFetch () composable function, which is Nuxt.js default function to fetch data from an API.

await useFetch('END\_POINT\_OF\_AN\_API');

NUXT 3 currently does not support axios.

Nuxt can fire the useFetch () function inside the <script setup>…</script> on both the server and in the browser (think about universal rendering here).

We will be using fake store API to fetch data in useFetch composable function.

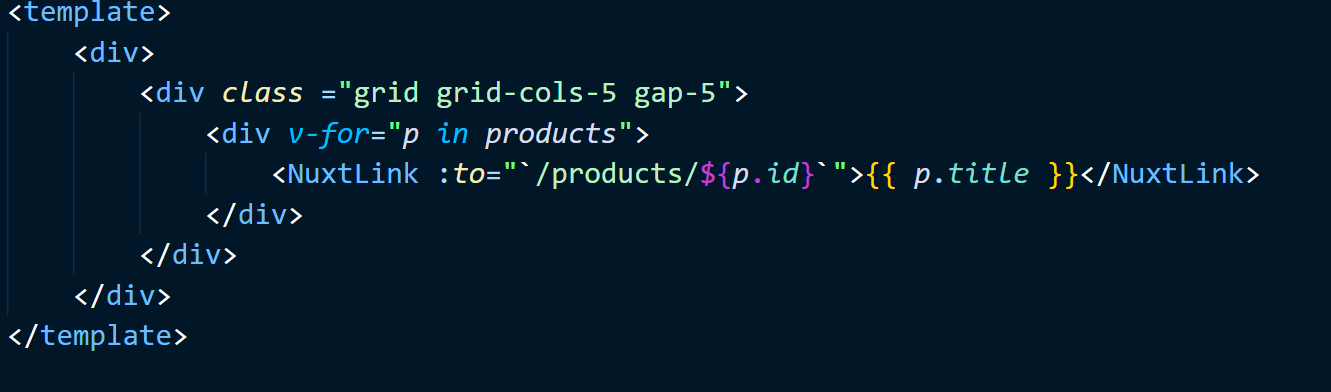
[https://fakestoreapi.com](https://fakestoreapi.com/)

1. Fetching All Products:

To fetch all products, we will be writing code in pages/products/index.vue file.

*const {data:products} = await useFetch(*'https://fakestoreapi.com/products'*)*;

here we are de-structuring the response we get from API end point. We get data from the end point but we want to use name as “products” in the code so we rename data as products using a “:” in de-structuring, and we can now use “products” in our template.





1. fetching a single product:

to fetch a single product from an API, we first get the id from the URL using useRoute().params;

*const {id} = useRoute().params*;

then we make an End point of the API dynamically using id that we got from useRoute().params;

*const uri =* 'https://fakestoreapi.com/products/'*+*id;

now we use the await useFetch('END\_POINT\_OF\_AN\_API'); composable function to send http request to the API.

*const {data:product} = await useFetch(*uri*,{key:*id*})*;

notice here that we have an extra parameter in useFetch() that is *{key:*id};

we use this key:id to tell useFetch() to call the api request each time there is a route change and fetch unique data.

Now in the template we can print the data like this:

<template>

  <div>

<p> {{*product*.*title*}} </p>

<p> {{*product*.*price*}} </p>

  <p> {{*product*.*id*}} </p>

</div>

</template>

**Creating Custom Components:**

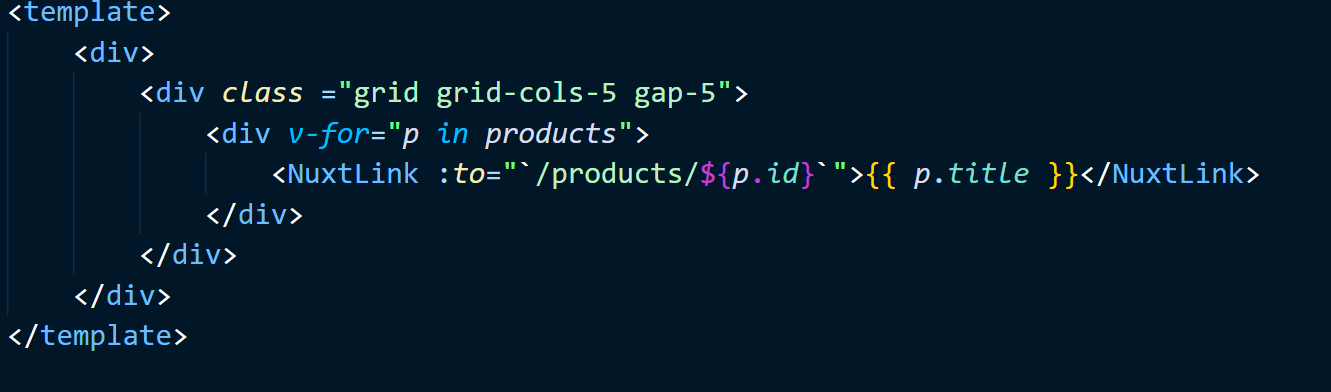
Create a folder in the root directory called components. When we create a component in this folder, we will not need to import this component as NUXT automatically import this for us.

In our case we want to display the products in localhost:3000/products in a card display. So, for that we will create a new file inside components folder named ProductCard.vue file.

Now, in pages/Products/index.vue file we are using

<NuxtLink :to="`/products/ ${*p*.*id*}`">{{ *p*.*title* }}</NuxtLink>

To move to the product detail page (localhost:3000/products/id)



Now, we want to call a component in pages/Products/index.vue file <template> so that we can implement the card view of the data.

So, we will remove the <NuxtLink> in <template> and put < ProductCard /> as component in it. Note here that we will not need to import this to use it.

Now, as we are looping through the products response, we got from useFetch() in pages/Products/index.vue file, we will pass each product (“p”) to < ProductCard /> component as props. We will name our props as :*productData. We are using “:” to pass props dynamically.*

<template>

  <div>

<div *v-for*="*p* *in* *products*">

  <ProductCard :*productData*="*p*"/>

</div>

</div>

</template>

Now, in components/ProductCard.vue file, we need a way to receive the props in the component. We receive props in a component, the following way:

*const {productData} = defineProps([*

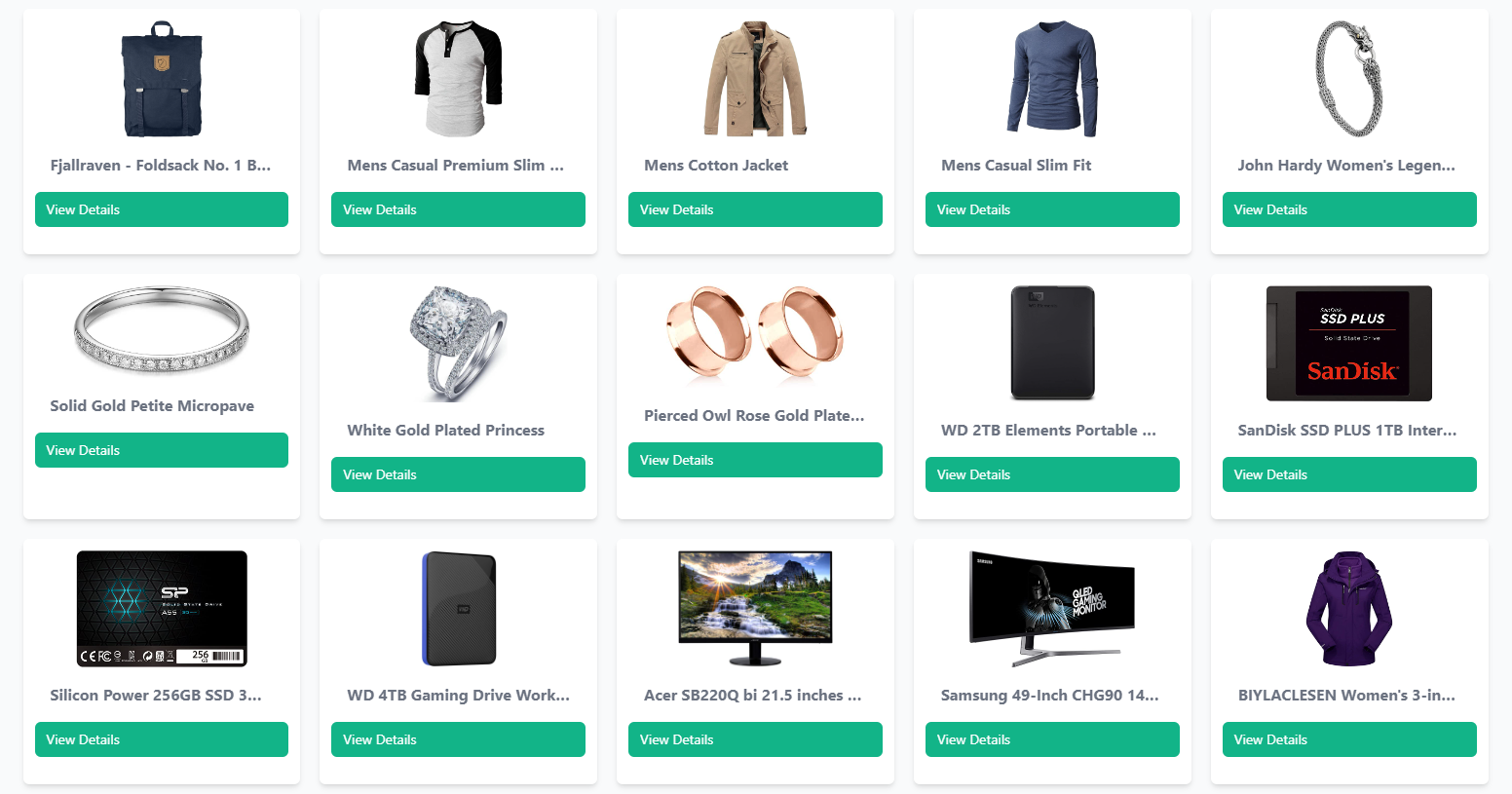
'productData'

*])*

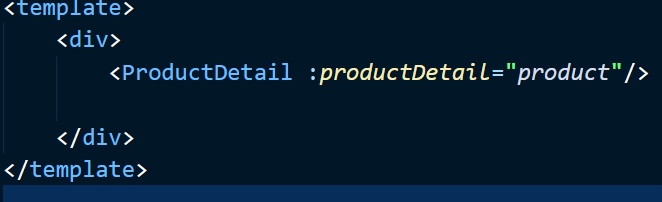
Now we have access to all the data that came from API response in *productData*. We now use *productData* inside the template to display the products in cards. Here we are using tailwind CSS for styling, which we will not discuss in detail.



Now when you run the project you will see the following:

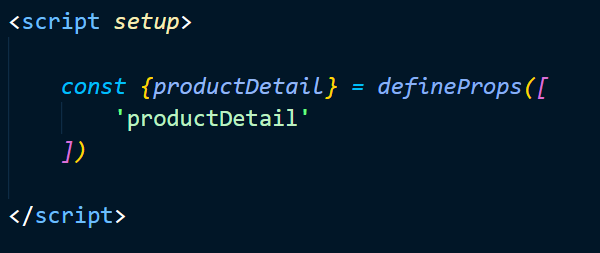


Now, to make a component to display individual product, we create a new component inside the component/ ProductDetail.vue file. Now, link this component to pages/products/[id].vue file and pass the response that we got from API in products as props.

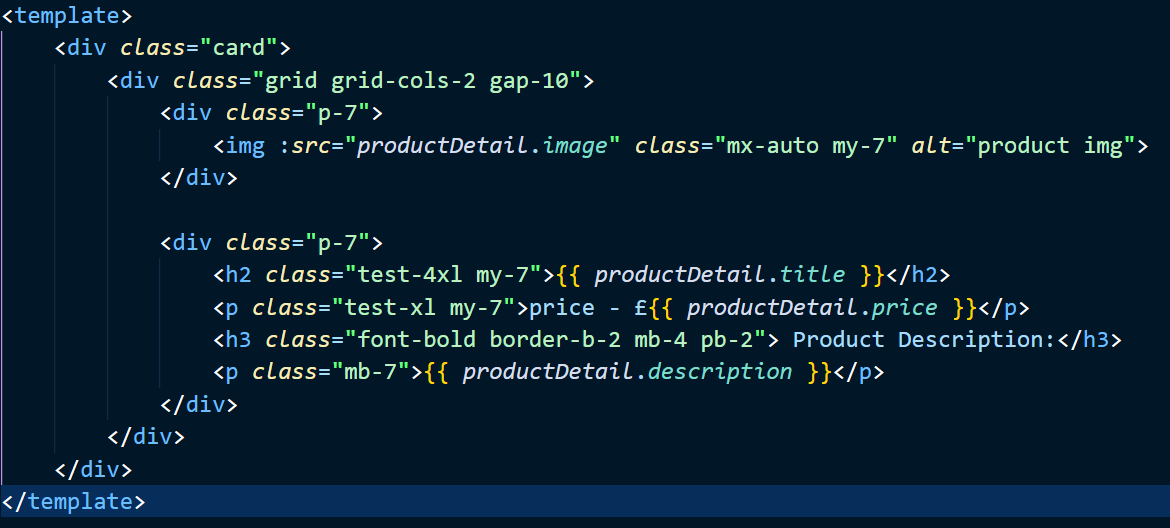


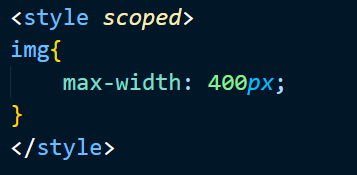


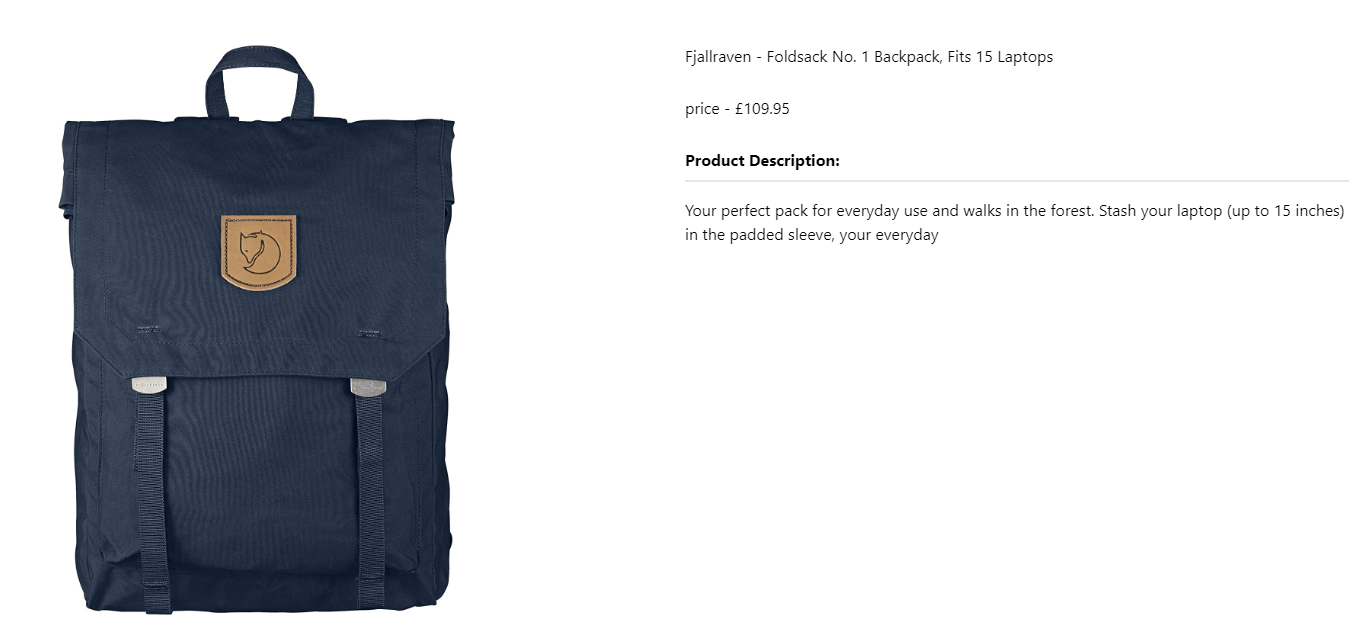
Now inside component/ ProductDetail.vue file, receive the props



Now use productDetail in template to display data in style using tailwind CSS.

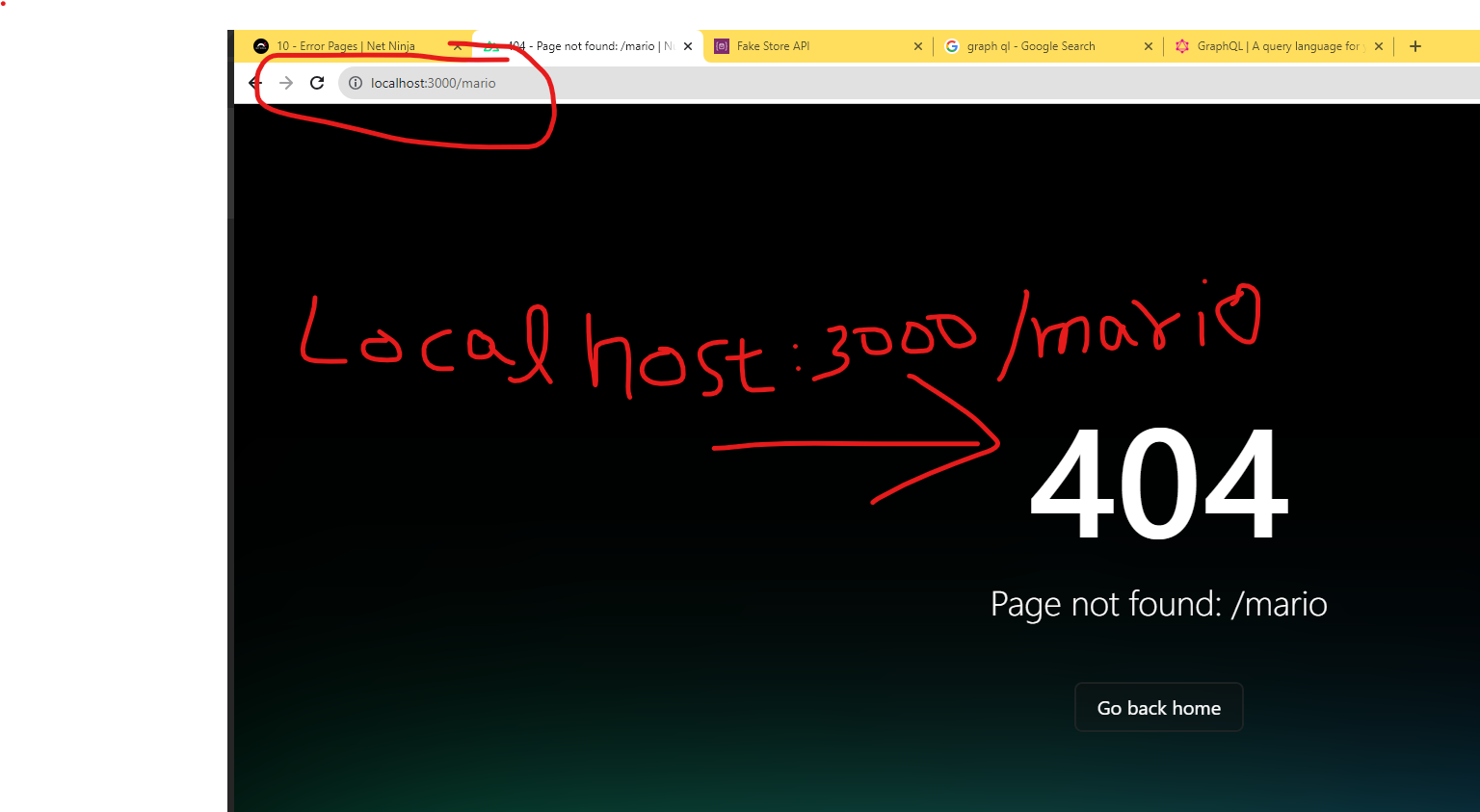




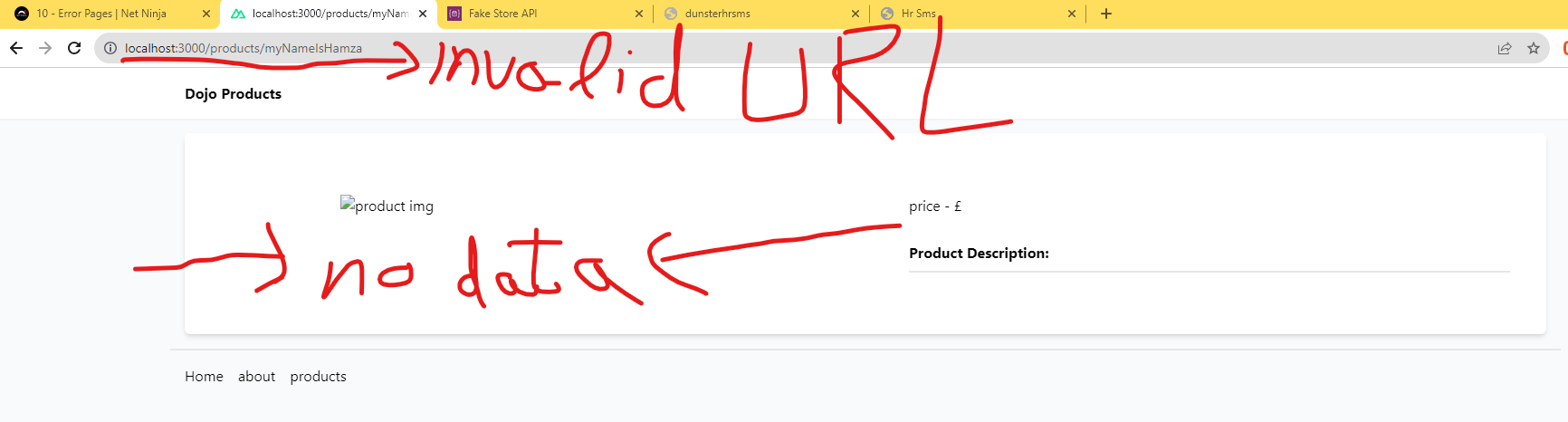


**Handeling Unexpected URL entered in address bar of browser**

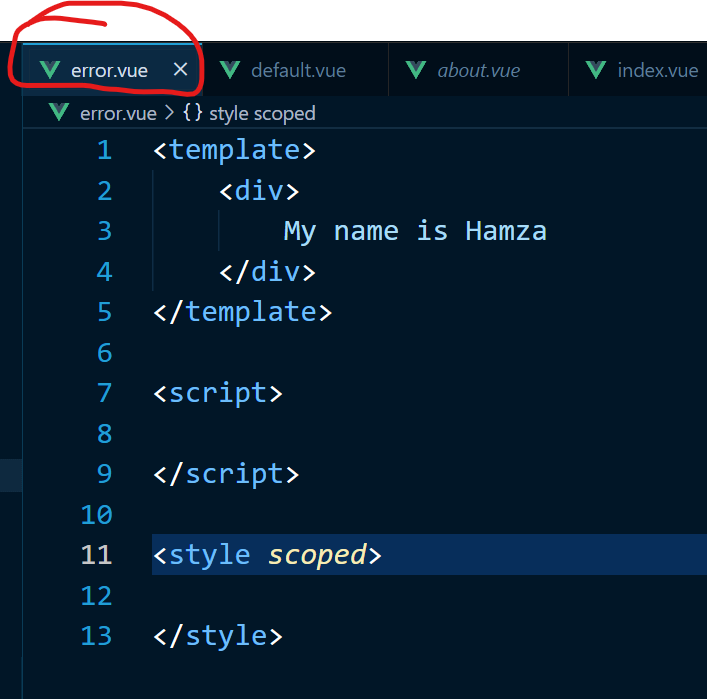
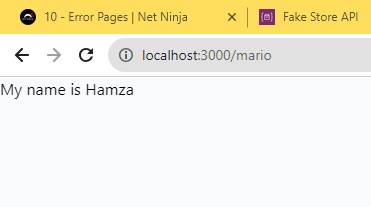
Currently, when we enter an unexpected URL (http://localhost:3000/mario) in the address bar in the browser, we get NUXT default 404 page.



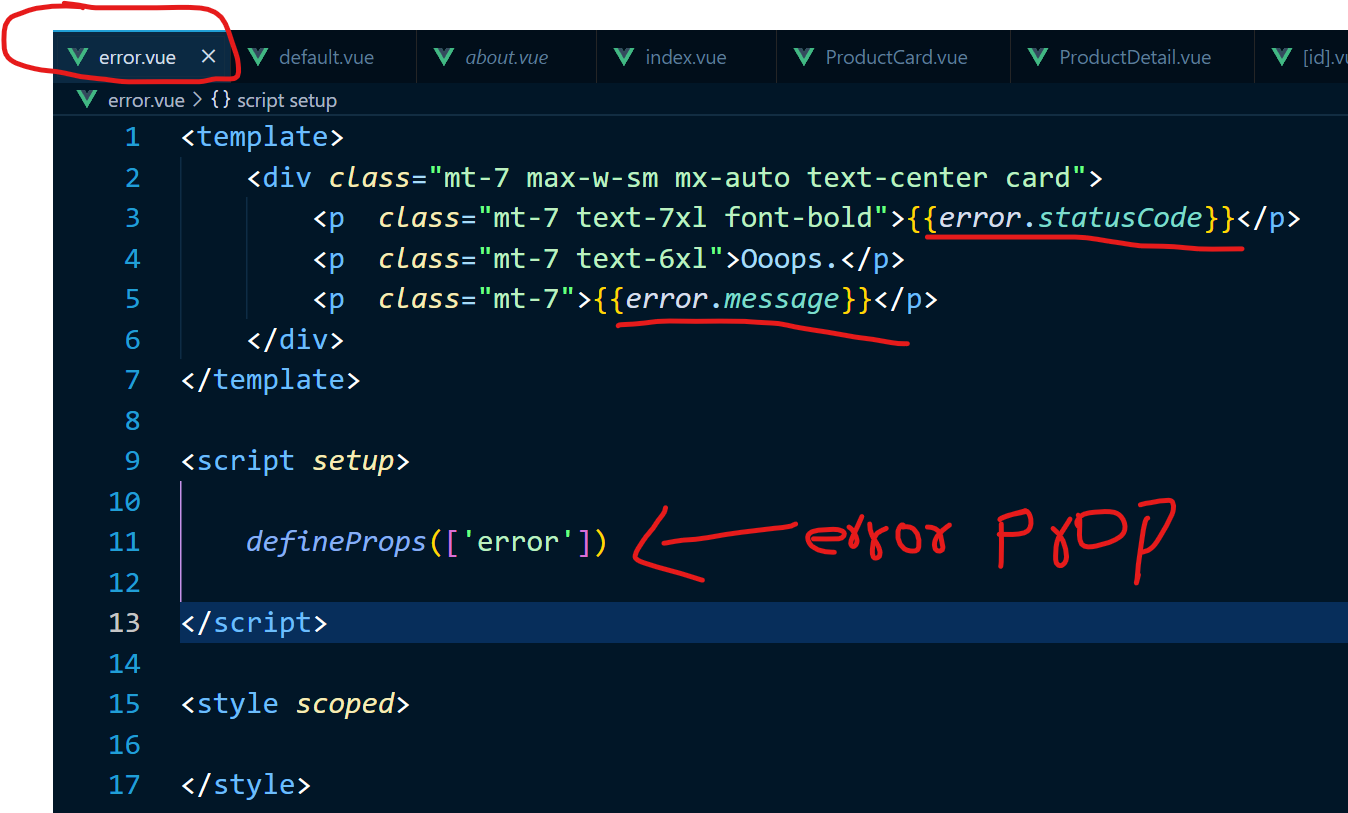
And also, when we type in invalid id of the product in the address bar of the browser (http://localhost:3000/products/myNameIsHamza), we still see the product page with empty data, but We should Not display the page to the user if we don’t have any data to display. So, we want to show a 404 message to the users if there is no data to be displayed or if they type in invalid URL.



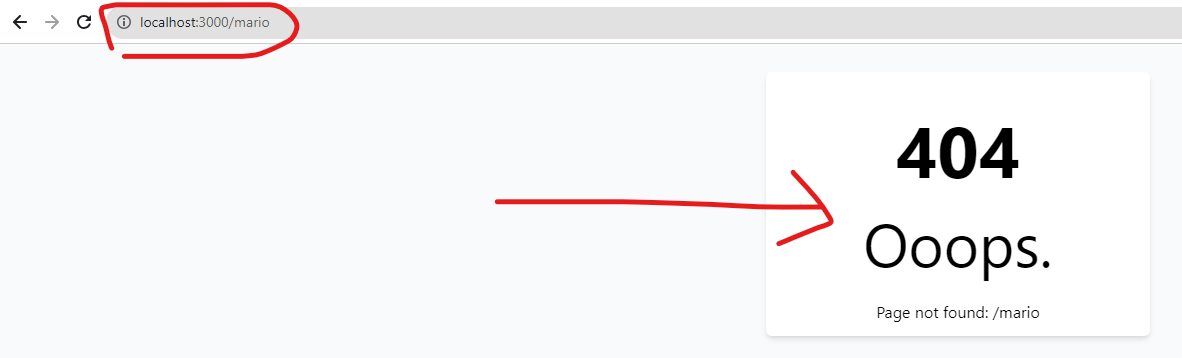
To solve this issue, we override the default NUXT 404 page and create our own custom error page. To create a custom error page, in root directory, just create a new file named error.vue, now we will see this file on browser, whenever an error occurs on the web page. So, if we now enter an invalid URL in address bar of browser (<http://localhost:3000/mario>), we will see the following.

But to actually display 404 error in our custom error page, we will need to accept a prop in <script setup> </script> tags. This Prop is the error prop given to us by Nuxt by default. We can then use this error prop in template to display error status code and error message as shown below.

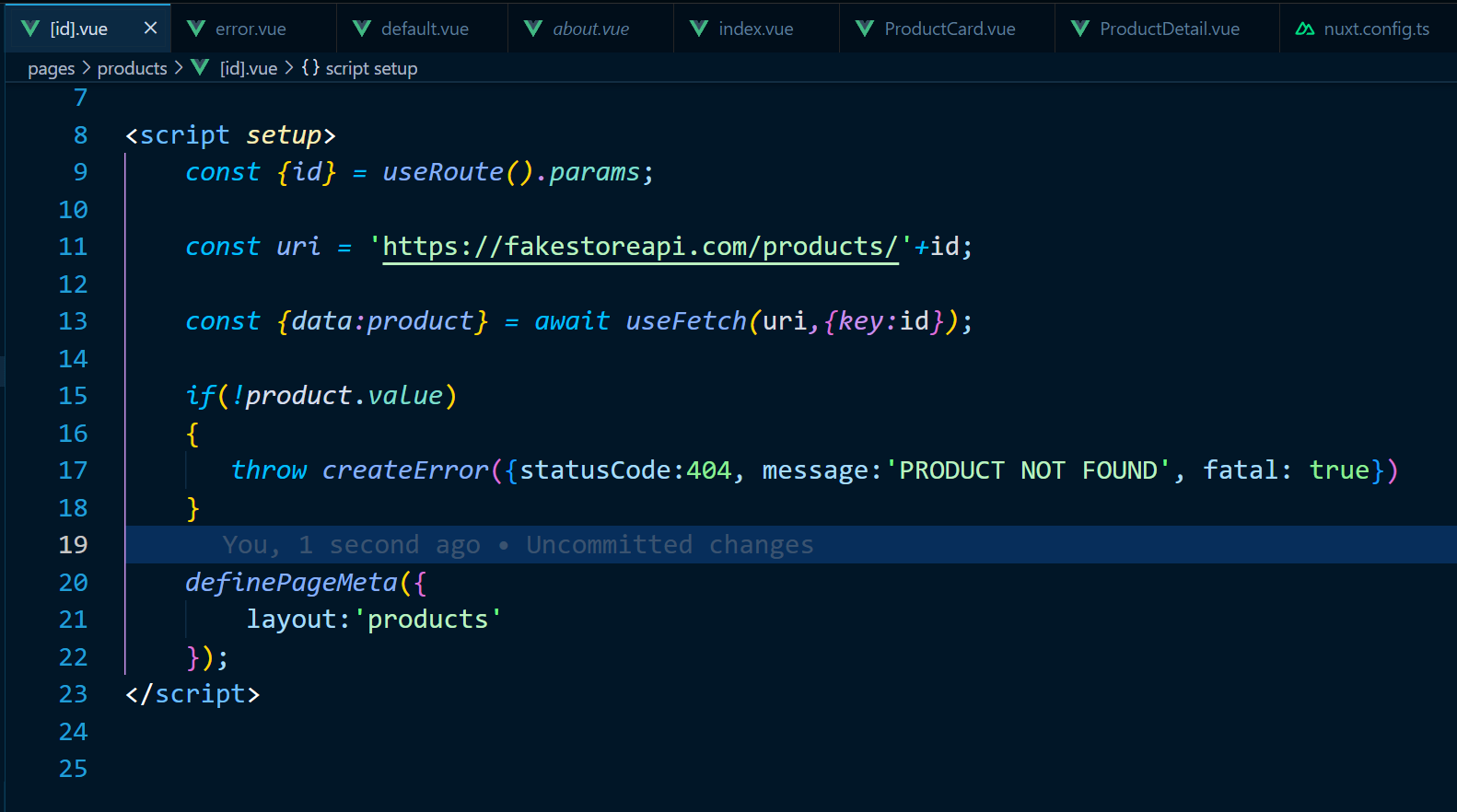


So now if we go to (<http://localhost:3000/mario>) we will see the following displayed.

****

**But we still have some issues to be solved here, for example, even after implementing the error page, if we go to** <http://localhost:3000/products/myNameIsHamza> we will still see this page displaying with empty data, but we want to show 404 error page on invalid ID of product (myNameIsHamza ).

Now to achieve this. We now go to pages/products/[id].vue file, and write a check for the API response, that after we get the API response, if the API response is empty, we will throw an error.

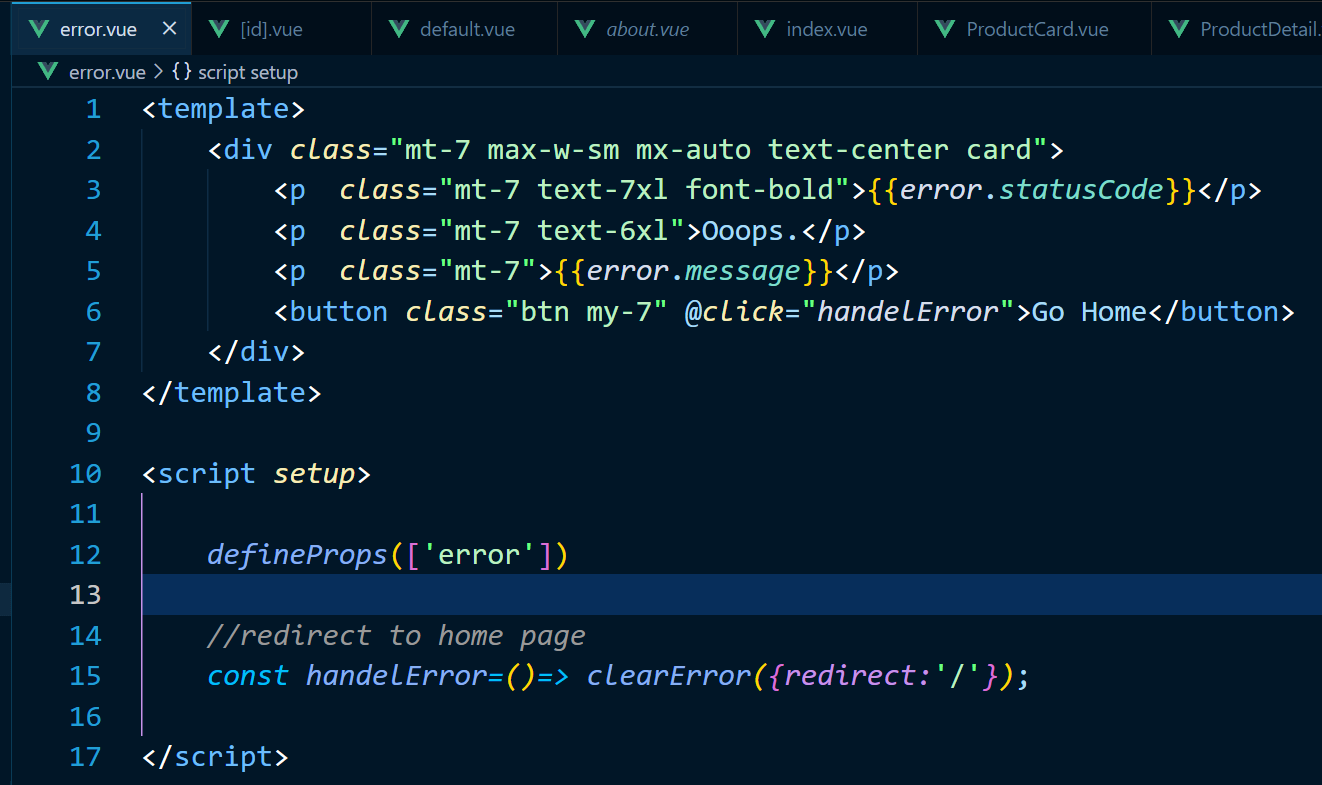


Nuxt gives us a default function to create an error called createError ({}), which accepts an object as an argument. We can pass in the status code and message here. What ever status code or message we pass in createError ({}) function, will be received as props to error.vue file in root directory, which displays all the errors to the screen.

NOTE: we use fatal: true here because, normally all our routes are rendered on server side, and if there is any error, we will be displayed 404 page, fine. But when we have a browser route, it does not display 404 page, so we are forcing NUXT to display 404 error page on all routes

But now if I want to add a button to the 404 page, so that when the user clicks on it, they go back to the home page.

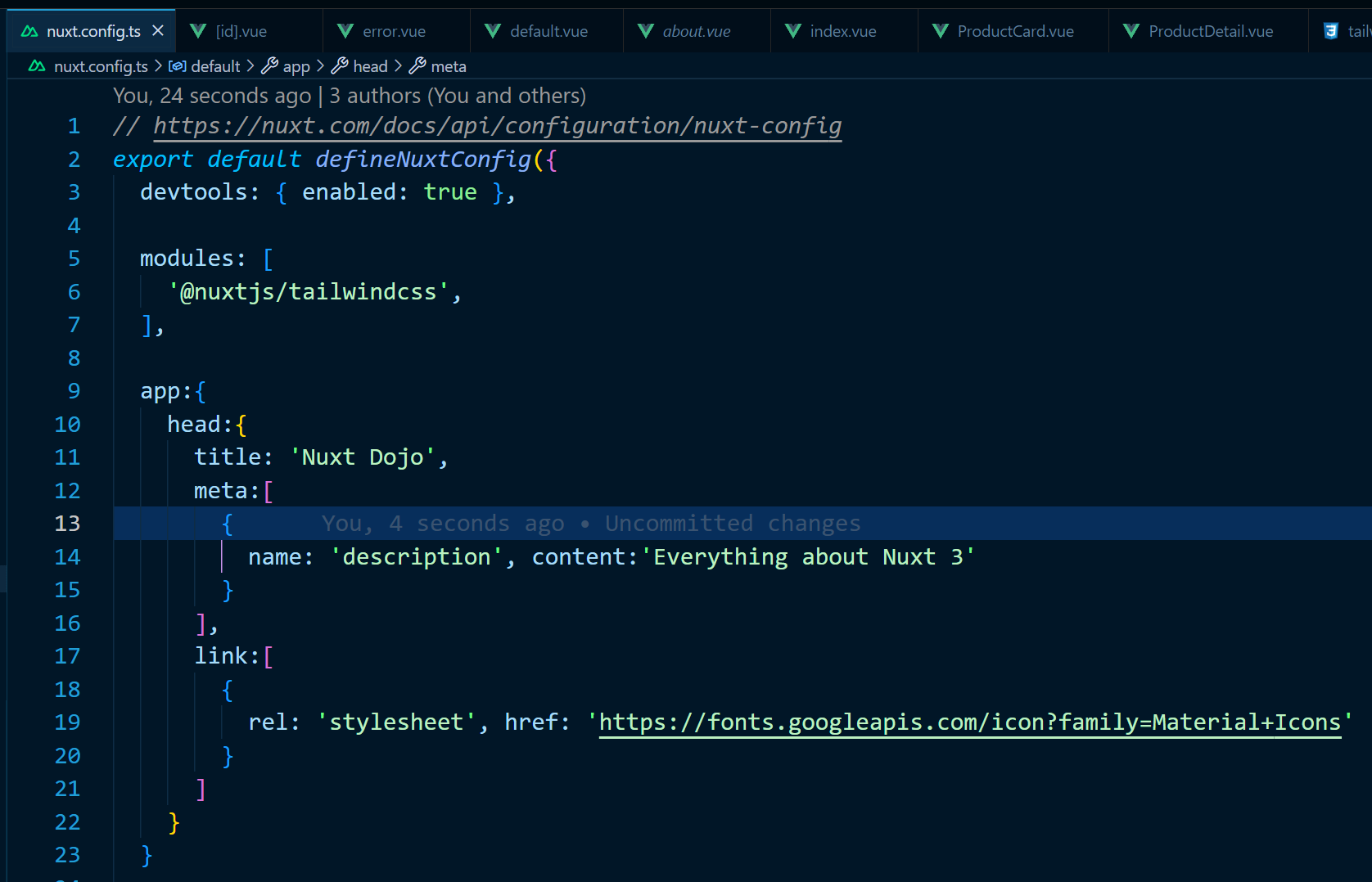
To do this we create a button in template in error.vue file, and call a function @click. Now this function is implemented inside the script tags, which will in return call a default NUXT function clearError ({}) to clear error and redirect to the path we give it.



**Metadata and useHead ()**

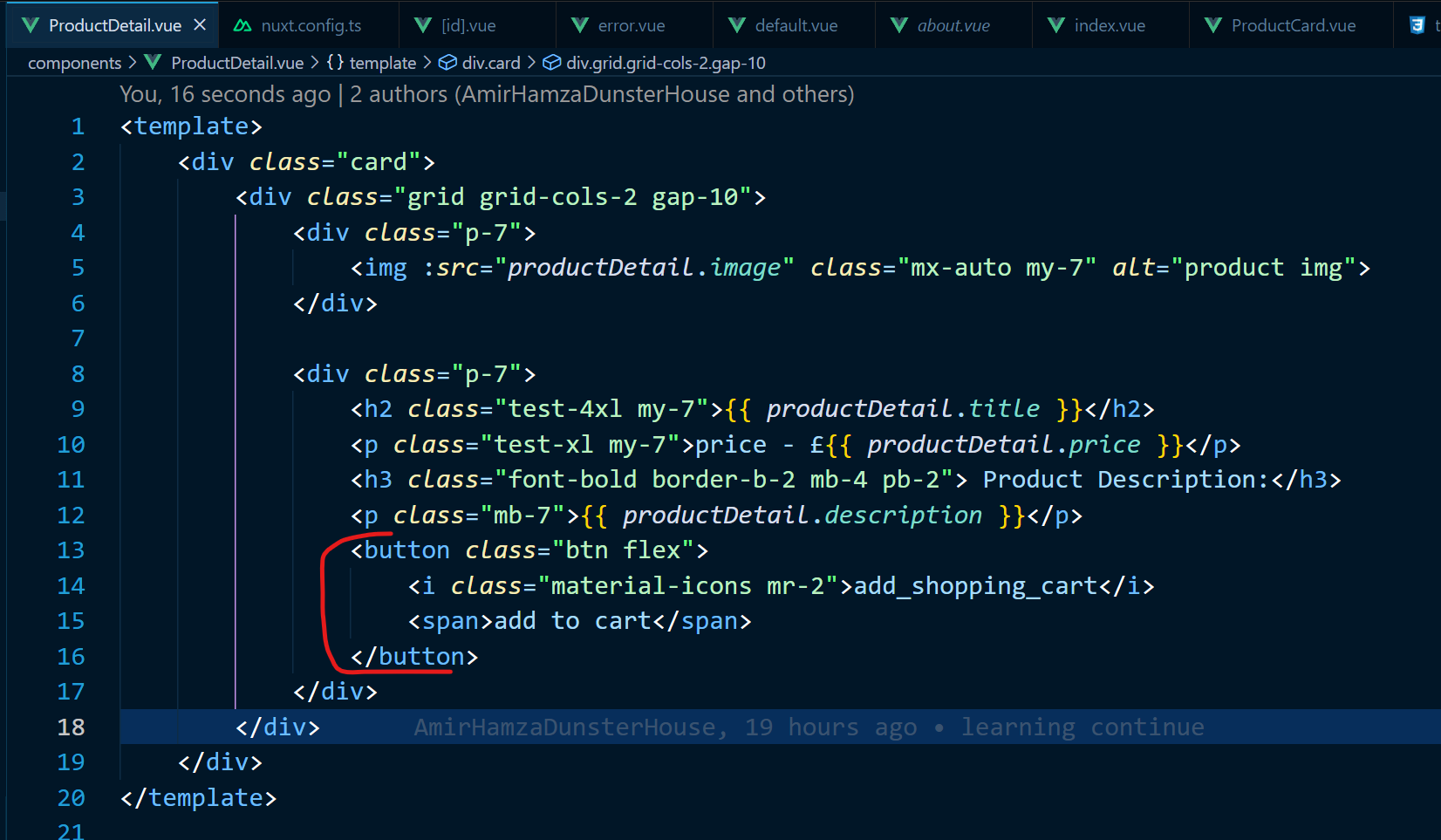
We will be discussing about Metadata of the website in this section, things like title, Meta tags for site description, document head, links to external style sheets etc.

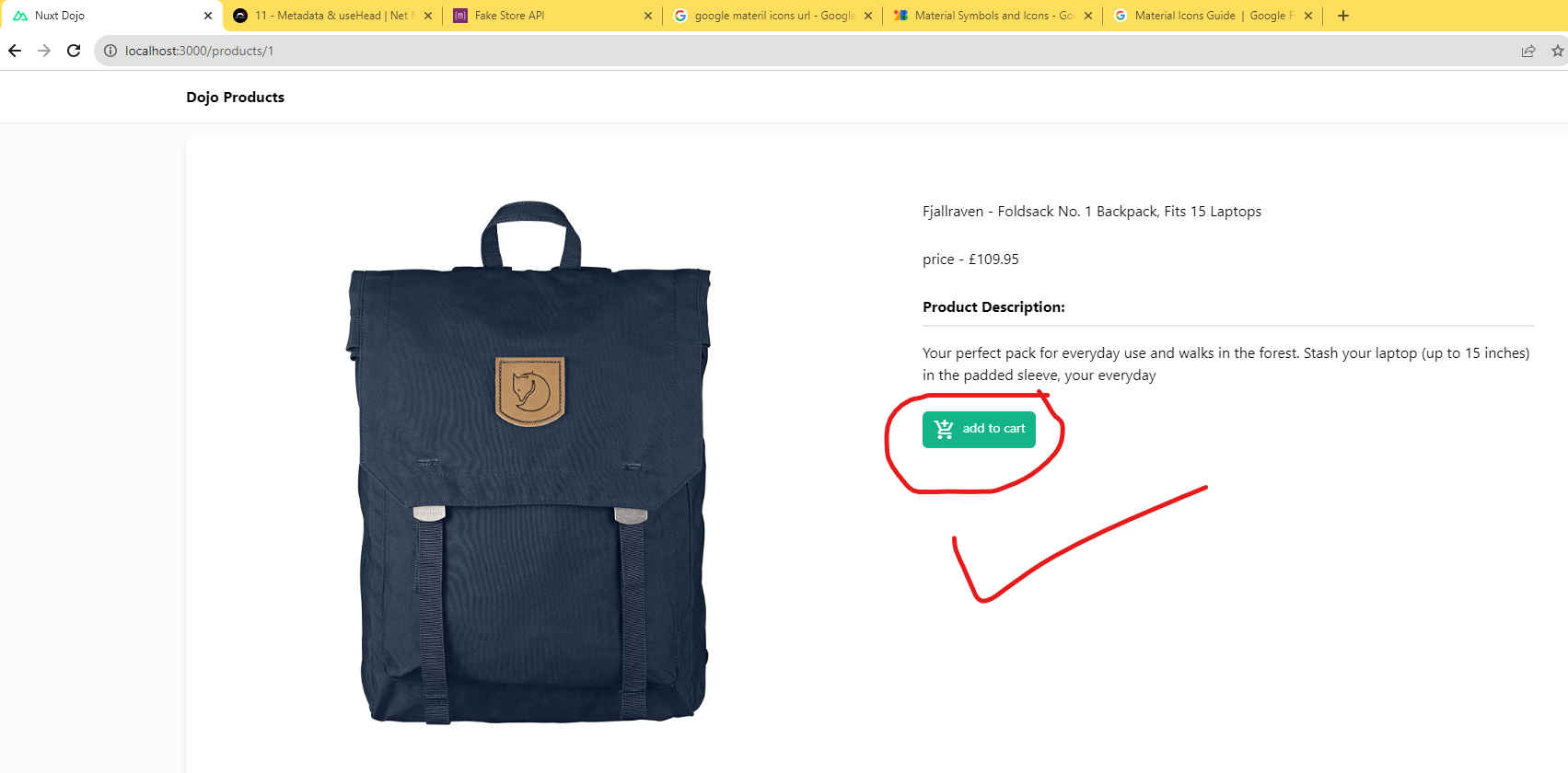
To add all the <head></head> tags content to our project, which is displayed on all pages of the site, NUXT has given us nuxt.config.ts file. We can define our Metadata as shown below



As we can see in the image that, after the modules array, we are adding app: {} object. Head: {} is nested inside the app: {} object. In head: {} we define our title, meta tag description, and link to google material icons. All this information will be available on all the pages of the website.

Now if we go to components/ProductDetail.vue file and we add a button at the bottom, to add the product to the cart, we will need to implement cart icon from google material icons. We can do this as shown in the code below. We now have access to google material icons because we added the link to google material icons in the nuxt.config.ts file.





As, the above method to add meta data is globally available to the whole website, so the same Meta data and title is applied to every single page of the web application.

But what if we want to override the default metadata or give a different metadata to every single page of the web application?

To achieve this by using useHead({}) composable function, to override the default metadata.



The other way to achieve this is by using NUXT components in <template> as shown below:

In Pages/products/index.vue file

<Head> <! --I like this way easy-- >

     <Title>Nuxt Dojo | products</Title>

     <Meta *name*='description' *content*="Nuxt 3 products page"/>

</Head>

And in Pages/products/[id].vue file:

<Head>

     <Title>Nuxt Dojo | {{product.title}}</Title>

     <Meta *name*='description' :*content*="product.description"/>

</Head>

**Custom Server Routes:**

**Dynamic Custom Server Routes:**

**Deploying to Netlify:**