**Next.js Tutorial**

## Contents

Introduction 2

Setup 3

* File Structure 3
* Flow of control 4

Routing 5

* Nested routes 6
* Dynamic routes 7
* Nested Dynamic routes 8
* Catch All Routes 9
* Link Component Navigation 10
* Navigating Programmatically 13
* Custom 404 page 14

### **Introduction**

A picture containing background pattern

Description automatically generated

What is Next.js

-The React Framework is for building production-ready applications

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

### **Setup**

Install/update node.js

<https://nodejs.org/en/>

In VS code, run the following command to create a next application:

npx create-next-app ‘app name here’

CD to the newly created directory with the application name and start a development server with ‘yarn dev’.

**File Structure**

Text

Description automatically generated

Here is our new project. Lets look at the files.

**package.json**

This file contains the dependencies and scripts required for the project. As new dependencies are added through yarn or npm, they will be added to the lists here. New scripts can be created and placed in this file as well.

**yarn.lock or package-lock.json**

Depending on whether you have used yarn or npm as a package manager, this file essentially ensures consistent installation of dependencies and does not need to be edited - don’t worry about this file.

**.gitignore**

This is a file relating to version control and we know what it does

**README.md**

In this case this readme gives information about next.js, its various uses etc.

**next.config.js**

This is the next.js configuration file which only has one module active within it when the project is created - ‘reactStrictMode’ which is set to true. This is a development mode only feature which is for highlighting potential problems within an application. It helps to identify unsafe lifecycles, legacy API usage and a number of other features.

**.eslintrc.json**

This is the config file for eslint. The code within this file, when it is initially generated, relate to a bunch of rules put together by the next.js development team.

**.next folder**

This is the folder that is generated when we run the dev or build scripts. Text

Description automatically generated

It is from this folder that our next.js application is ‘served’.

**node\_modules folder**

This is the folder in which all the dependencies are installed. It is generated when you run the command yarn/npm install. In our case, it was generated when the yarn dev script was executed which installs all the dependencies required if they’re not present.

**styles folder**

This folder contains some styles for our project. It is possible to define styles in a few other folders within the project so this folder does not have real significance. CSS basically can go here.

**public folder**

This folder holds all the public resources for our application. Images, svgs, icons etc. This folder is different to the similar one created in a React application. In that scenario this folder would contain an index.js file which would contain the single html page for the react application. There are no files of this type in this file in next.js. This folder is for public resources.

**pages folder**

This folder is responsible for the entire routing feature in a next.js application. This will be covered in detail in this tutorial.

**index.js**

This file is the one that is served in the browser when we visit the local host.

**\_app.js**

This is where we can define the layout of our application

**api folder**

This is where we can create apis for our application.

**Flow of control**

When you type yarn dev or npm run dev the execution goes through the following steps.

1. package.json where the script is recognised.
2. \_app.js

Text

Description automatically generated

This file contains the ‘MyApp’ component. This will automatically receive the Component and PageProps as props which are then returned as part of the jsx.

Index.js



When running the application on the development server, the component prop will refer to the component defined in the index.js file which in our case is the Home component.

### **Routing**

Text, letter

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

In this tutorial we’ll focus JUST on the filing system and how the routing works. So delete the api folder and the index.js file both of which are in the pages folder. Leave public and styles folders as they are not relevant.

As mentioned in the introduction, next.js has a file-system based router. The convention is that when a file is added to the pages dir/ it is automatically available as a rout.

Graphical user interface, application

Description automatically generated

With page-based routing, lets add a rout that needs to be rendered when the user visits our website (localhost:3000). To achieve this, we create a file called index.js in the pages folder. Create a function in this component and give the return statement a h1 tag with ‘Home page’. Then run yarn dev and this should load a page with the words ‘Home Page’ displayed.

**The index.js file located in your pages folder will map to the root of your domain.**

**Graphical user interface, text, application

Description automatically generated**

Now lets add the above routs for our domain. This is a simple case of creating .js files for each with a simple function exporting.

Text

Description automatically generated This is an example.

**Pages are associated with a rout based on their file name. E.g. about.js maps to localhost:3000/about**

**Nested Routes**

**Graphical user interface, application

Description automatically generated**

Here we have a scenario where we want access to a blog page but then access the first blog or second blog from there.

Part 1 of this would be completed the same way as the example in scenario 2. How to implement the nested routes for first and second blog. Very simple. This can be easily achieved in next.js using nested folders in the file structure. So we create a blog FOLDER and put a first and second.js file in there to create the nested components. Now we have a blog folder with two files in it but the original blog.js file is still in the pages folder. It would be nice to be able to put all the files relating to one domain area together right? This is very simple to do. We just move the blog.js file INTO the blog folder and rename it as index.js. **An index.js file inside a folder, nested within the pages folder, will map to the rout of the folder name as a rout.**

**Dynamic Routes**

Here we will build a product and listing page and then an individual page:

Graphical user interface, application

Description automatically generated

The first part here is achieved in much the same way as scenario 2. As you can see, using the scenario 3 process would require creating an individual .js file for each product. Obviously this is not plausible for a store with thousands of products. The correct solution is to use DYNAMIC ROUTE SEGMENTS. In next.js you need to add square brackets to a file name to create a nested dynamic route. Here we’ve created one called [productId].js. Pascal case is being used here. Inside this file we simply create the same function that we have done previously…

Text

Description automatically generated

Now we can type ‘http://localhost:3000/products/2’ or even ‘http://localhost:3000/products/12430’. This is possible because next.js treats square brackets in a file name as a dynamic segment to create a dynamic route. So now we have created a dynamic route but now we need to improve the jsx. At the moment we are only displaying ‘product details go here!’ irrespective of which product we have chosen. In a typical application we would want to extract the id of a product and display something to do with that specific product on it’s corresponding page. Perhaps by retrieving information from a db by making an api call using the product id. In this simple scenario let’s just display the product ID in the browser. In order to do this we need to import a hook from the next.js package. This hook is called useRouter and returns a router object. From this object we can access the query object - router.query. The parameter we want to access is productid 'router.query.productId’. It is important to note here that productid on the query object corresponds to the dynamic segment we have specified for the file name. This needs to be the same. Now we can access the productid in the jsx.

Text

Description automatically generated

It is important to note that as part of the router.query.productId is actually a string so any word or number or combination could be placed in there. Lets add the word ‘sweater’ to the url - <http://localhost:3000/products/sweater>. As before it will load up a page with the product details of {productid} here which is ‘sweater’. If you then create a new nested route called ‘sweater’ that loads say a string ‘Loading page for sweaters’, next.js will FIRST search through the route folder for a file with that name and display that as a nested route. If it cannot find the specific file name THEN it will load the dynamic nested route instead. Cool functionality.

**Nested Dynamic Routes**

Graphical user interface, application

Description automatically generated

Here we will explore how to display a product by it’s ID as in the previous scenario but then display a specific review for that product. A nested folder structure is the answer here as in scenario 3. In this case though, the nested folder will have square brackets. As before, we want the productid.js functionality to be stored within this folder as with scenario 3 so we move it into there and change it’s name to index.js. Now that we have this nested folder set up (and it’s dynamic properties are still working as expected because of the inclusion of the square brackets) we can now create a new NESTED dynamic folder WITHIN this folder. This is then essentially following scenario 4. Here we do not have to put the folder ‘review’ into square brackets because it does not have any more nested dynamic routes. Were we to add another nested level, lets say [http://localhost:3000/products/2/review/21/**comments/130**](http://localhost:3000/products/2/review/21/comments/130)**,** then we would have to repeat the same process as we have done here in scenario 4 for the new nested dynamic routes.

Text

Description automatically generated

You can see here that the router.query is assigned to more than one variable so you need to use destructuring to access the different variables - see line 5.

**Catch All Routes**

Graphical user interface, text, application

Description automatically generated

So here the example shows a webpage where there are a number of features and each feature has a number of corresponding concepts. We can see that if there are 20 features each containing 20 concepts then all of a sudden we’ve got 400 files that would be necessary to create. We’ve discovered in scenarios 4 and 5 that dynamic routing is possible so we could incorporate this to have each feature having dynamic nested concepts which would leave us needing to make 20 files. We could do one better still and have a dynamic feature and a dynamic concept which would mean only creating 1 file. In order to make this possible next.js has special file convention that is used:



Here we see that the params file has both the square brackets denoting a dynamic structure but also the ‘…’ three dots similar to the spread operator. The name params is a convention and it refers to the parameters that are passed into this file. Within the file lets create a simple component. Graphical user interface, text

Description automatically generated

What is special about this page is that it will match any url that contains the docs segment in the path. So for example <http://localhost:3000/docs/feature1/> and <http://localhost:3000/docs/feature1/comment1> will both return the […params].js component.

<http://localhost:3000/docs/anything-you-like-after-the-word-docs>.

**This catch all route catches all the URL segments and maps them into one single file in the project.** This is useful for something like a documentation site because we want the different segments in the URL for better organisation and SEO but the layout for the document will remain the same. We’ve seen above in scenario 5 how to access multiple variables within the router.query using destructuring. The same applies here through the params keyword - it could be any word but this is the convention. In the example above where we have <http://localhost:3000/docs/features/1/comments/27> then the params constant would return an array with 4 elements: features, 1, comments, 27.



Instead of returning a string, params returns an array of strings.

It’s also worth noting that initially ‘undefined’ is returned. This is because of the prerendering feature in next.js which will be covered later in this tutorial. As a quick fix you could set the initial value of the params to an empty array to avoid a runtime error.

Text

Description automatically generated

Now lets update this file to display more information relating to the parameters passed in. Here we use if statements to work through the length of the array and act according to the array length.

A screenshot of a computer

Description automatically generated with medium confidence

This is a simple setup and ideally you would have each if statement fetching specific documents from the db via an api.

Graphical user interface, application, website

Description automatically generated

Here’s an example of the system working. Scenario 6 has been implemented now.

Another example use-case could be a property website where the user wants to view houses BUT they also only want to view the ones that are within a budget of 250k and 500k. Something like: <http://localhost:3000/propertywebsite/houses/250000/500000>. The values could easily be accessed using the params process set out in scenario 6 and then used to create an array of the houses that fall within that category.

It is also worth noting that next.js provides optional catch-all routes. Were we to type <http://localhost:3000/docs> we would currently get a 404. These optional routes allow us to deal with this and will be covered later in this tutorial. In simple terms it can be delt with by rapping the params file in a second set of square brackets.

### **Link Component Navigation**

Typically there would be a UI element the user would click in order to navigate to a different route or the user could be navigated programmatically after a specific action has completed. Lets explore this.

For the first example, using the file structure and files already created from the above tutorials relating to routes, let’s try navigating from the home page to the blog page. To do this we simply use the Link component. This is for client-side navigation. Firstly import the component in the index.js file inside pages. This is the root index file for the app. Then simply create a link tag with the href of where you want the link to go to:

Text

Description automatically generated

The anchor gives the visual link to press like a button and the href tells the link where to take you to. You can do the same in the destination page to tell it to go back to the home page. The href in this case is simply “/”.

Navigating to dynamic routes is also relatively easy. You could create a link for each of the items:

Text

Description automatically generated But you can see here that this will very quickly become unsuitable and we have a dynamic range here so the number could be anything from 3 to 3 million… A more appropriate solution would be to pass the product id as a prop to the component. Text

Description automatically generatedText

Description automatically generated

Also something to keep in mind is the ‘replace’ prop which can be added to the link div. A screenshot of a computer

Description automatically generated with medium confidence

When pressing back from a page that has been linked to in this manner the url will reset to the home page.

**Navigating Programmatically**

Graphical user interface

Description automatically generated with medium confidence

The scenario here is that on occasion it will be necessary to programmatically navigate to a new page after an event has occurred. Here we have a place order page and on the click we want to place the order which will return us to the product page. Perhaps a similar example for my application would be the programmed navigation that would follow a new user signing up to the service and then being sent to their new profile page.

The logic will all take place in the onClick handler function and the useRouter hook is the key here. One of the many router methods is router.push() which takes in the destination url as it’s argument. It has other useful parameters as well which may be useful down the line. We’re passing in the products href but any href that has been used could be used instead. This includes dynamic and nested routes. There is also router.replace which replaces the history instead of pushing the route onto the stack. This will make sense later.

Custom 404 page

In any next.js application, if you navigate to a page that does not exist, a default 404 page will be loaded instead. If for any reason you want to create a custom version of this all you need do is place a 404.js file into the pages folder.

Graphical user interface, text

Description automatically generated

Text, letter

Description automatically generated

API - Application Programming Interface

Package Manager: yarn/npm

SEO - Search Engine Optimisation