Routing

All routes must be placed inside the app folder.

Every file that corresponds to a route must be named page.jsx or tsx.

Every folder corresponds to a path segment in the browser URL.

A screenshot of a computer

Description automatically generated

This page component will be rendered by navigating to **localhost:3000/about** URL.

The page.jsx file placed right under **app** directory will be the root page.

**Static Nested Routing**

**A screenshot of a computer

Description automatically generated**

Now entering **localhost:3000/about/aboutnested** in the URL will navigate to the new page.jsx component under the **aboutnested** directory.

**Dynamic Routing (Query Parameter)**

**A screenshot of a computer

Description automatically generated**

You can also implement dynamic routing with query parameter, meaning that the content of page will vary depending on the id value in the URL.

**localhost:3000/about/{*Id value*}**

To access the **Id value** in the URL,

A screen shot of a computer

Description automatically generated

**params** is a fixed name by convention, and aboutId is the **name of the dynamic routing folder** your page.jsx component is located in.

A screenshot of a computer

Description automatically generated

A screen shot of a computer program

Description automatically generated A screenshot of a computer

Description automatically generated

It can be nested too, and the values can be accessed like this

**Catch-all segments**

When the nested query in the url gets too complicated, we can use catch-all segments. It is useful for document page implementations.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screen shot of a computer program

Description automatically generated

The segs can be named whatever, but it must contain **…** characters before the name.

**Not Found Page**

Not Found page which users are redirected by entering invalid URL address can be made under any directories, as many as needed. But the name of the page has to be **not-found.jsx** by convention.

To navigate to the Not Found page programmatically you should import **notFound** function from “**next/navigation”.**

**A computer screen shot of a program code

Description automatically generated**

In this example if the value of **aboutId parameter** is higher than 1000, the user will be redirected to the not-found page.

A screenshot of a computer

Description automatically generated

A screen shot of a computer code

Description automatically generated

A screen shot of a computer

Description automatically generated

In this directory structure, the not-found page will be the default not-found page for about/{aboutId} page component.

**Private Folders**

It creates a folder which should not be included in the routing logic, by including underscore character at the beginning of the folder naming.

Used for separating UI logic from routing and consistent organizing of internal files across a project (such as utility files).

If you want to include an underscore in URL segments, you can prefix the folder name with “%5F”, which is the URL encoded form of an underscore.

A screenshot of a computer program

Description automatically generated

This page component will not be accessible by routing

**Route Grouping**

When you want to group different associated page components in a single folder (for example, login, register, forgot-password components can be grouped in the auth folder), we use route-grouping syntax.

A screenshot of a computer screen

Description automatically generated

By wrapping the folder name with parenthesis, you can exclude the filename **auth** from the URL, and the URL to the login page will be localhost:3000/login, instead of localhost:3000/auth/login

**Layout Page**

Layout component wraps the page component in the same directory, providing UI for the page component.

The Layout.jsx component in the root component will wrap the root page.jsx page, which will provide global UI components like navbar (just like app.jsx or main.jsx)

A screen shot of a computer code

Description automatically generated

The {children} is the child component of current component, in this case it is the page.jsx component in the src/app root directory.

It also can be nested by creating another layout.jsx file in sub-routing folder.

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generatedThe layout.jsx file

A screen shot of a computer

Description automatically generatedThe page.jsx file

A screenshot of a chat

Description automatically generated

Rendered result

The layout file can be used for grouped components too.

A screenshot of a computer

Description automatically generated

In this case the layout will be applied to the login and password page, but not on the signout page.

**Metadata**

Export a static metadata object

Export a dynamic generateMetadata function

**Rules**

Both layout.jsx and page.jsx files can export metadata. If defined in a layout, it applies to all pages in that layout, but if defined in a page, it applies only to that page.

Metadata is read in order, from the root level down to the final page level.

When there’s metadata in multiple places for the same route, they get combined, but page metadata will replace layout metadata if they have the same properties.

A screen shot of a computer code

Description automatically generated

You can access the URL query to be displayed on title too.

A screen shot of a computer screen

Description automatically generated

Typescript example

**Defining API Routes**

Unlike vanilla react projects you can define restful API routes within the same project when you work with Next.js

A screenshot of a phone

Description automatically generated

Under src/app directory, create a folder named **api .(name is fixed by convention), and** another sub folder named of your choice**,** and a js or ts file named **route.ts (fixed by convention too).** In the route.ts file you can define the function with names of only **GET, POST, PUT, DELETE.**

**GET request**

A screen shot of a computer program

Description automatically generated

Access the query params by using req.nextUrl.searchParams.get(***query name***).

**Navigating/Linking**

When you want to navigate to other pages in the application, **NEVER USE <a href=”#”> or window.location.** As these reload the entire project instead of loading just the requested components, it terminates the main purpose of using SPA(single page application), making the loading much slower.

Instead use <Link> component provided by ‘next/link’, or useRouter hook provided by next/navigation.

**import Link from 'next/link'**

**<Link href="/dashboard#settings">Settings</Link>**

**OR**

****

****

****

**To get the current path the user is in, use usePathname() hook**

**A computer screen shot of a program

Description automatically generated**

Display different components depending on the current path

**Redux**

**Redux** is global state management library for react. In other words it is like useState hook, but can be used globally throughout the project. It is commonly used for storing user’s credential or tokens, or any information that could be accessed commonly from any parts of the projects. In our example it could be the recipes, as the fetching of recipes takes couple of seconds to be completed.

**Implementing Redux**

**Step1: Create the slice and action**

**A screen shot of a computer program

Description automatically generated**

**Line # 2) initialUserInfoState** is like the declaration of state variable in react. It is equivalent to

const [userInfo, setUserInfo] = useState({userNickname:””, userIngredients: []}); using useState hook.

**Line # 3) Creating slice** is like reducer hook in react. You are defining functions that updates the values in the state.

**Line # 4, 5)**You are naming your slice which will allow you to access and update the state in future, and giving the state initial values.

**Line # 6~12)** Defining reducer functions. Taking setUserNickname function as example, it takes two arguments which are state and action (**NAMES ARE FIXED BY CONVENTION).**

This function is used to update the userNickname value in the state.



You are accessing the userNickname state by using the state parameter, and replacing with action.payload value, which is the parameter value passed in by the user.

If the user is not going to pass in any parameter for updating the value (for example, creating a function that clears the userNickname value in the state), the **action** parameter is not necessary.

At the end of the file you need to export the reducer value of the slice you created as default (line # 16), and actions of UserInfoSlice too.



**Step2: Create the store**

A screen shot of a computer program

Description automatically generated

To use the reducer functions and slices we created in step 1, you need to put them in a store variable and export it as default.

**Step 3: Wrap the components with Provider**

**A screen shot of a computer code

Description automatically generated**

Import the **Provider** component from “**react-redux**” and the **store** you created and exported in step 4. Pass the store variable to the store attribute of Provider component, and make the Provider tags wrap the components which you want to use these state value in. In this case I want to use the redux functionalities throughout the entire project, so I wrapped the root component.

**Step 4: Access/Update the states**

To **access** the value in the state, you need to import the **useSelector** hook from “react-redux” package.



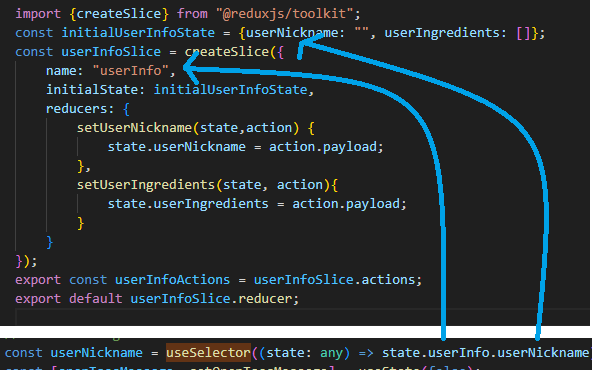
And store it in a variable at the start of a component function, just like other useState declarations.

A screen shot of a computer

Description automatically generated

Although the userNickname variable is declared as const, the value will change accordingly because it is a state, not a fixed string value.



This line means that you want to access the **userInfoSlice** with the name “userInfo”, and the userNickname state of the slice. 

After declaring the userNickname const state, you can use it anywhere in the function like a state value, and the value will be shared throughout the entire project.

To **update** the state value, you need to import the **useDispatch** hook from “react-redux” package, and the Actions that was exported in **step #1.**





A screen shot of a computer screen

Description automatically generated

In this logout function, the value passed to the setUserNickname is the **action.payload** in the reducer function. We are setting the userNickname state in the slice to empty string, deleting the user credential.



**Making Requests**

**A screen shot of a computer program

Description automatically generated**

Unlike in the GraphQL/React mixmate project implementation by Ulises where all requests made to the server are fixed to POST, we can make all GET, POST, PUT, DELETE requests to the server. This makeRequest function defined in the utility class handles the requests with different request functions more easily.   
  
  
  
  
**Sample Usage of makeRequest function**

**A screen shot of a computer program

Description automatically generated**

**API\_ROUTES.user = ‘/user’**

**REQ\_METHODS.post = ‘POST’**

**newUserInfo =**

**A computer screen shot of a computer code

Description automatically generated**

The request function will process the data passed in which is the newUserInfo in this case and make the POST request to the API route “domain/api/user” to save a new user on the database.

**JWT Authentication/Authorization**

The two most common way of validating user credentials are session authentication and token authentication. We are using token authentication in our scenario using **jsonwebtoken** library.

**Step 1. Generate token**

****

**A screen shot of a computer code

Description automatically generated**

(Api/login/route.ts/POST Function)

Tokens are generated using the **sign()** function from the jwt package. It takes in 3 parameters,

**Payload :** Usually it contains the entity about the user or additional data such as the datetime the token was issued.

**Secret key :** It is a random string that is used to generate the encrypted token.

**Options:** Optional parameter that can include various settings.

* **algorithm**: Specifies the algorithm to be used for signing the token (e.g., **HS256**, **RS256**). If not specified, the default is **HS256**.
* **expiresIn**: Defines the time for the token to expire (e.g., **60**, **"2 days"**, **"10h"**).
* **notBefore**: Defines a period before which the token is not acceptable (**nbf** claim).
* **audience**: Intended recipient of the token (**aud** claim).
* **issuer**: The issuer of the token (**iss** claim).
* **jwtid**: A unique identifier for the token (**jti** claim).

**Step 2. Return the token in the response header**

The **token.token** in cookie is just a string separated by 3 periods.

A screen shot of a computer code

Description automatically generated

**api/login/route.ts**

The **token.token** in **cookie** string is just a string separated by 3 periods.

Instead of manually including the token in the result object which contains the userInfo data, it will be included in the response header called ‘Set-Cookie’. Through this step the token will be stored in a safe storage called httpOnly cookie, which is never accessible from the client side with javascript codes. After this step the token will be included in any http requests by default when user send any request to the API routes.

The client side does not need any additional codes to handle the returned token, since it will be automatically stored in the httpOnly cookie.

**Step 3. Verify the token (client side)**

**A computer screen shot of text

Description automatically generated**

**(pages)/(auth)/login/page.ts**

Execute the makeRequest function to send a POST request to /verifytoken API route. The data parameter can be just an empty object, since the token will be automatically included in the request header.

**Step 4. Verify the token (server side)**

A screen shot of a computer program

Description automatically generated

**api/verifytoken/route.ts**

This is the GET function of **verifytoken** API route.

A computer screen with text

Description automatically generatedThe function starts by grabbing the token string from the header.

A computer screen shot of text

Description automatically generated

Using the **verify** function from the jwt library, the token will be verified if it is authorized and valid.

If the token is not valid it will throw errors to be handled.

A screen shot of a computer code

Description automatically generated

Finally return the result of verification.

**Step 5. Delete the token**

A screen shot of a computer program

Description automatically generated

A black background with white text

Description automatically generated

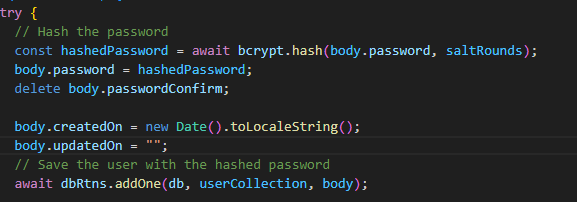
The comment in the POST function explains everything lol

**Password Hashing**

It is always necessary to store the password on the database in encrypted format, instead of plaintext format.

**Step 1. Sign up new user**

****

****

**Api/user/route.ts/POST function**

The **body object** contains the user information such as username and password sent as a request from the client.

And the password in the body object will be hashed with **hash()** function from the **bcrypt** library.

**Step 2. Log in**

**A black screen with white text

Description automatically generated**

**A screen shot of a computer code

Description automatically generated**

**Api/login/route.ts/POST function**

**Body** object is the object included in the request sent by the user which contains the **plaintext** nickname and password, and **userInfo** object is the object fetched from the database.

To compare if the password values match, we use the **compare** function from the **bcrypt** library. **Make sure the plaintext value is passed in as the first argument, and the hashed value is passed in as the second argument.**

The returned value from the bcrypt.compare function will always be Boolean.