Build a Posts App with NextJS

Knowledge Requirements:

1. VisualStudio Code (Or any other Editors you prefer)
    a. https://code.visualstudio.com/
2. TypeScript (knowledge of any other static typing language ok)
    a. https://www.typescriptlang.org/docs/
3. React (functional components, hooks & JSX)
    a. https://reactjs.org/tutorial/tutorial.html
4. Minimum Database structures and how they work (SQL Queries?)
    a. https://www.w3schools.com/sql/
5. How REST API's work
    a. https://www.cleo.com/blog/blog-knowledge-base-what-is-rest-api

Other Requirements:

1. NodeJS
    a. https://nodejs.org/en/
2. Yarn
    a. https://classic.yarnpkg.com/lang/en/docs/install/#windows-stable
3. Docker(docker-compose)
    a. https://www.docker.com/products/docker-desktop/
4. Prisma
    a. https://www.prisma.io/docs/concepts/components/prisma-cli/installation
    b. https://www.prisma.io/docs/getting-started/quickstart
5. Next-Auth
    a. https://next-auth.js.org/getting-started/introduction
6. ChakraUI
    a. https://chakra-ui.com/guides/getting-started/nextjs-guide

Setting up the environment

1. Install NodeJS – Link in requirements
2. Install Yarn
   npm install --global yarn
3. Create a NextJS App
   yarn create next-app –typescript
4. Set up your database
   Create a docker-compose.yml file containing:

```
version: "3"
services:
  mysql:
    image: "mysql"
    restart: always
    environment:
      MYSQL_DATABASE: postsv1
      MYSQL_ROOT_PASSWORD: mypassword
    ports:
      - "3306:3306"
    expose:
      - "3306"
    volumes:
      - "./.data/db:/var/lib/mysql"
volumes:
  db-data:
    driver: local
```

   In the root folder tun the command / docker-compose up
5. Install Prisma - `yarn add prisma –dev`

   `Prisma commands example:`

```
Set up a new Prisma project
  $ prisma init
  Generate artifacts (e.g. Prisma Client)
  $ prisma generate
  Browse your data
  $ prisma studio
  Create migrations from your Prisma schema, apply them to the database, generate
artifacts (e.g. Prisma Client)
  $ prisma migrate dev
  Pull the schema from an existing database, updating the Prisma schema
  $ prisma db pull
  Push the Prisma schema state to the database
  $ prisma db push
```

6. Initiate the prisma project
   yarn prisma init

   A prisma/schema.prisma file will be generated inside the root folder this file will be used to generate de database schema further.

   In order to continue inside the .env file you need to add a DATABASE_URL Environment variable that will contain the connection string to your local db created previously using docker-compose

   ConnectionString composition = “databaseType://user:password@address:port/database” in our case the Environment Variable will look like this:

```
DATABASE_URL="mysql://root:mypassword@localhost:3306/postsv1"
```

7. Inside the schema.prisma we will be creating our db objects/models
   Example:

```
model Posts {
  id String @id @default(uuid())
  name String
  description String
}
```

The above model from schema.prisma will be represented as a Table – Posts with the columns: id, name, description where id will be the key of the table with a default unique id generated on each record.

8. Generating the Database content
   In order to generate the content we need to install the prisma client and then generate the client and migrate the content

   - yarn add @prisma/client
   - yarn prisma generate
   - yarn prisma migrate dev

   After the migrate command you will be asked to provide a name for the migration.
   Now your database contains a table named Posts with the defined fields.

   If you want to access and view your data and tables from your db you can use Prisma Studio

   If you get a wrong credentials notification it means you are running another service on port 3306 (either stop that service or run this one on another port)

NextJS Introduction

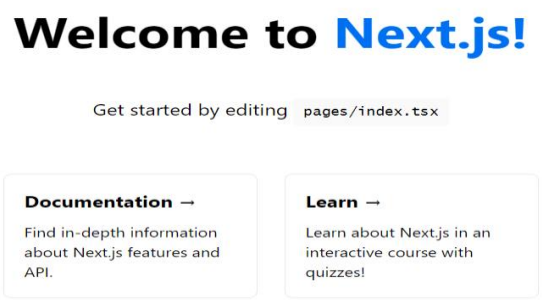
1. Running the project

In order to run the project in the terminal we will use the command – yarn dev

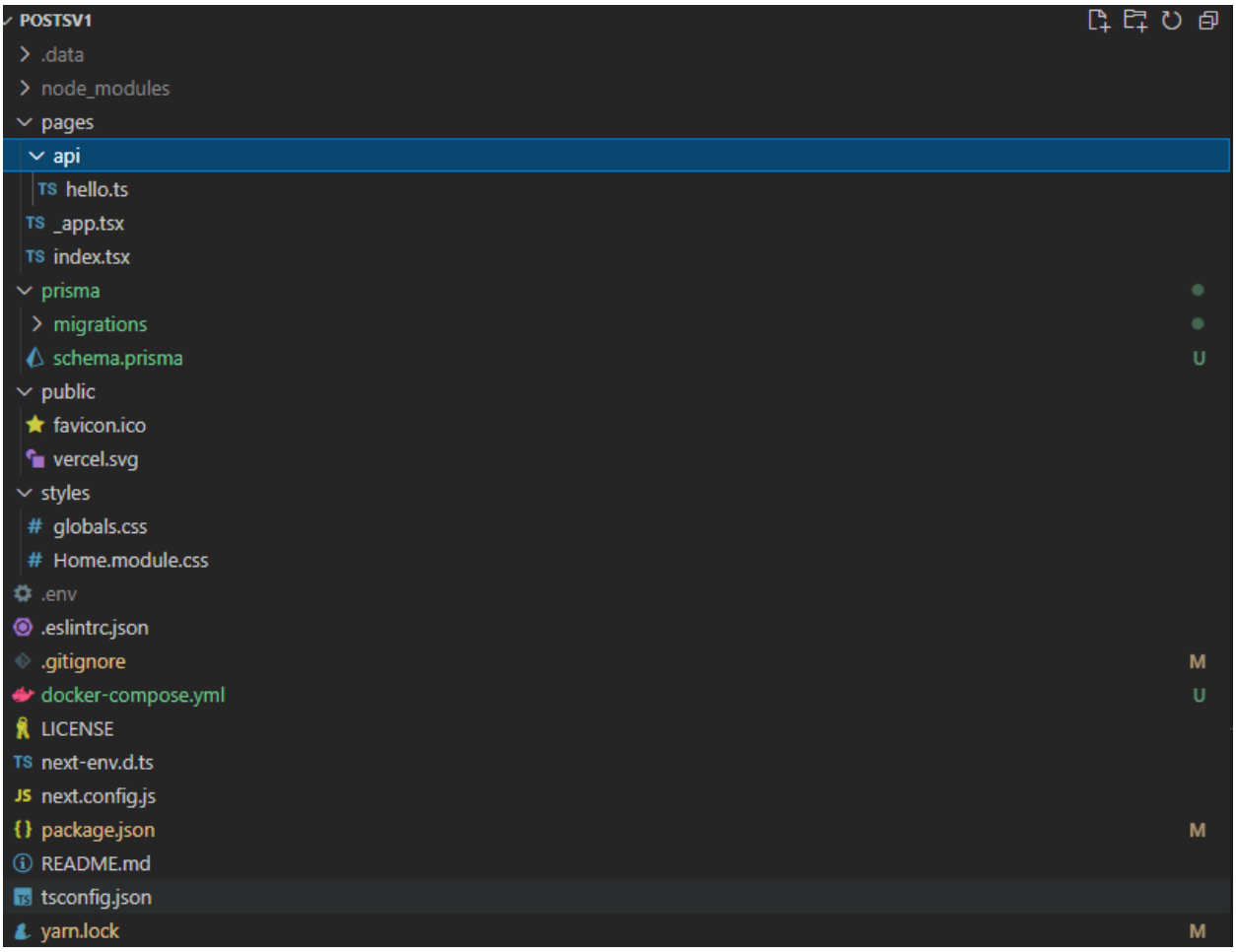
This will run the application on the default port 3000 on your localhost

http://localhost:3000/

Accessing the above link will render your application:



2. Project structure at this point



When initially created the project will contain only the folders: pages, pages/api, public, styles

NextJS provides routing out of the box. Every route under the /pages directory will also be the route accessed in the browser

For example:

http://localhost:3000/ will run the index.tsx file under pages (./pages/index.tsx)

http://localhost:3000/api wil not run anything due to the lack of the index.tsx file inside the directory

http://localhost:3000/api/hello will run the “. /pages/api/hello.ts”

Every .tsx file inside the pages folder will be considered as a page for your application

API endpoints are added in pages/api

The current structure provides you with an endpoint example.

```
// Next.js API route support: https://nextjs.org/docs/api-routes/introduction
import type { NextApiRequest, NextApiResponse } from "next";

type Data = {
  name: string;
};

const handler = (req: NextApiRequest, res: NextApiResponse<Data>) => {
  res.status(200).json({ name: "John Doe" });
};

export default handler;
```

3. Endpoint structure

Now, the example they provide works with any method. Either a GET or a POST will return the same response with a status 200 and a json containing {"name": "John Doe"}

In order to treat this case we will need to use a switch() statement or if() statements.

An example would be:

```
// Next.js API route support: https://nextjs.org/docs/api-routes/introduction
import type { NextApiRequest, NextApiResponse } from "next";

type Data = {
  name: string;
```

```
};
type Error = {
  error: string;
};

const handler = (req: NextApiRequest, res: NextApiResponse<Data | Error>) => {
  switch (req.method) {
    case "GET":
      res.status(200).json({ name: "John Doe" });
    case "POST":
      res.status(405).json({ error: "Method Not alowed" });
    case "DELETE":
      res.status(405).json({ error: "Method Not alowed" });
    case "PUT":
      res.status(405).json({ error: "Method Not alowed" });
  }
};

export default handler;
```

This option we can all agree is a little tedious. Using an if statement will look the same.

So.. In order to make things look more clean and have the possibility to treat all cases easily we can use “next-connect”.

In order to install it just run - `yarn add next-connect@0.12.2`

Now we can easily import “next-connect” into our application

An example of an endpoint using next-connect:

```
import type { NextApiRequest, NextApiResponse } from "next";
import nc from "next-connect";

type Data = {
  data: string;
};
type Error = {
  error: string;
};

//Define the GET Method
const getMethod = async (req: NextApiRequest, res: NextApiResponse<Data>) => {
  res.status(200).json({ data: "This is the Get Method" });
};
```

```
export default nc({
  onError(error: any, req: NextApiRequest, res: NextApiResponse<Error>) {
    res.status(501).json({ error: "Error" });
  },
  onNoMatch(req: NextApiRequest, res: NextApiResponse<Error>) {
    res.status(405).json({ error: "Method Not Allowed" });
  },
})
  .get(getMethod)
  //or you can add the method directly to the post action
  .post(async (req: NextApiRequest, res: NextApiResponse<Data>) => {
    res.status(200).json({ data: "This is the Post Method" });
  });
```

`You can treat errors or Non matching request methods by adding them in the next connect call.`

```
export default nc({
  onError(error: any, req: NextApiRequest, res: NextApiResponse<Error>) {
    res.status(501).json({ error: "Error" });
  },
  onNoMatch(req: NextApiRequest, res: NextApiResponse<Error>) {
    res.status(405).json({ error: "Method Not Allowed" });
  },
})
```

`You can either define the methods and use them in the nextconnect`

```
//Define the GET Method
const getMethod = async (req: NextApiRequest, res: NextApiResponse<Data>) => {
  res.status(200).json({ data: "This is the Get Method" });
};
export default nc({
  onError(error: any, req: NextApiRequest, res: NextApiResponse<Error>) {
    res.status(501).json({ error: "Error" });
  },
  onNoMatch(req: NextApiRequest, res: NextApiResponse<Error>) {
    res.status(405).json({ error: "Method Not Allowed" });
  },
})
  .get(getMethod)
```

Or if you prefer define them directly in the method call

```
  .post(async (req: NextApiRequest, res: NextApiResponse<Data>) => {
    res.status(200).json({ data: "This is the Post Method" });
  });
```

Now if you try to request a delete method it will enter the onNoMatch Block. Or on errors it will enter the onError Block.

Creating an App

For the sake of understanding how to use NextJS. We will start building a minimalistic app where users can add posts. We will not complicate things. As a general idea:

- User must be logged in to see Posts
- User can add, edit or delete their own posts.
- The database will contain only users, and posts table.
- The application will contain a login page and a posts page

First things first, we will define a Layout for the app. Let's say we need:

- Navbar
  - App Title
  - Dark/Light Mode
  - Sign In Button
  - Account Icon
    - Drop Down
      - Profile Picture
      - User Name
      - Log Out Button
- Body

All of this should be a Layout/Template for the current app, and based on what address is accessed the body should change accordingly.

For example, we will have a /posts link that will fill the body of the layout with Posts

Then we will need a Post component that will contain:

- Posts
  - Post Name
  - User Name
  - Creation Date
  - Post Description

Navbar

We will create a file **“navbar.tsx”** inside the path **“../components/Navbar”**

This file will contain all the JSX necessary to paint the Navbar.

navbar.tsx

```
import {
  Box,
  Flex,
  Avatar,
  Button,
  Menu,
  MenuButton,
  MenuList,
  MenuItem,
  MenuDivider,
  useColorModeValue,
  Stack,
  useColorMode,
  Center,
} from "@chakra-ui/react";
import { MoonIcon, SunIcon } from "@chakra-ui/icons";
import { NextPage } from "next";

const Navbar: NextPage = () => {
  const { colorMode, toggleColorMode } = useColorMode();

  const session = false;

  return (
    <>
      <Box bg={useColorModeValue("gray.100", "gray.900")} px={4}>
        <Flex h={16} alignItems={"center"} justifyContent={"space-between"}>
          <Box>PostsApp</Box>

          <Flex alignItems={"center"}>
            <Stack direction={"row"} spacing={7}>
              <Button onClick={toggleColorMode}>
                {colorMode === "light" ? <MoonIcon /> : <SunIcon />}
              </Button>

              {!session ? (
                <Menu>
                  <Stack direction={"row"}>
                    <Button
                      onClick={() => {}}
```

```
                    variant={"solid"}
                    cursor={"pointer"}
                  >
                    SignIn
                  </Button>
                </Stack>
              </Menu>
            ) : (
              <Menu>
                <MenuButton
                  as={Button}
                  rounded={"full"}
                  variant={"link"}
                  cursor={"pointer"}
                  minW={0}
                >
                  <Avatar size={"sm"} src={“image”} />
                </MenuButton>
                <MenuList alignItems={"center"}>
                  <br />
                  <Center>
                    <Avatar size={"2xl"} src={“image”} />
                  </Center>
                  <br />
                  <Center>
                    <p>{"userName"}</p>
                  </Center>
                  <br />
                  <MenuDivider />
                  <MenuItem onClick={() => {}}>Logout</MenuItem>
                </MenuList>
              </Menu>
            )}
          </Stack>
        </Flex>
      </Flex>
    </Box>
  </>
 );
};

export default Navbar;
```

Navbar notions explained

1. `const { colorMode, toggleColorMode } = useColorMode();`

This is a custom hook offered by “ChakraUI” that will allow us to change the color mode from Light to Dark in our application.

colorMode is a string of “light” or “dark”

toggleColorModel is a function iside the hook that will cycle through “light” or “dark”

2. `const session = false;`

This line will be changed with the current user session later when we will add Authentication in our app.

It will be used to cycle through the SignIn Button if a current session does not exist and the User Icon if the session exists.

3. `bg={useColorModeValue("gray.100", "gray.900")}`

This line tells the useColorMode hook that in light mode the navbar should be colored in gray.100 and in dark mode with gray.900. We need to clearly distinguish the navbar from the rest of the app so we override the default colors.

4. Color Mode toggle

```
<Button onClick={toggleColorMode}>
  {colorMode === "light" ? <MoonIcon /> : <SunIcon />}
</Button>
```

Here we have a Button which, if clicked will toggle the colorMode variable from “light” to “dark” and vice versa.

Inside the Button tags we have a ternary in which we verify if colorMode is equal to “light”

If the expression is true we will show the MoonIcon otherwise show the SunIcon.

5. SignIn Button or User Avatar

Immediately after the Color Mode Toggle you will see another ternary.

A simple explanation without going through the whole JSX is If there is no session the paint the SignIn Button otherwise Paint the Avatar of the User => { !session ? ShowSignIn : ShowAvatar}

Template

Since we now have a Navbar we must create a template that we will use for our web app.

This template should have the Navbar and a Body in which we will render our other pages

We will create a file **“template.tsx”** inside the path **“../components/Template”**

template.tsx

```tsx
import { StackProps, VStack } from "@chakra-ui/react";
import { FC, ReactNode } from "react";
import Navbar from "../Navbar/navbar";

export type LayoutProps = StackProps & {
  children?: ReactNode;
};

const Template: FC<LayoutProps> = ({ children, ...props }) => {
  return (
    <>
      <Navbar />
      <VStack
        flex={1}
        alignItems="center"
        justifyContent="flex-start"
        paddingTop="10"
        {...props}
      >
        {children}
      </VStack>
    </>
  );
};

export default Template;
```

Template notions explained

Since this is TypeScript, in order to pass children in a component we need specify to the function what kind of parameter it should get.

For that we will create a type for the current components

```tsx
export type LayoutProps = StackProps & {
  children?: ReactNode;
};
```

What this does is that it tells the function that it will accept an object containing the children inside that will exist inside the <Template></Template> tags.

Now we specify that Template is a Functional Component (FC) and that it should inherit the LayoutProps previously created

Inside the component we will return the Navbar and all the children passed in the Template Vertically Stacked

```
  return (
    <>
      <Navbar />
      <VStack
        flex={1}
        alignItems="center"
        justifyContent="flex-start"
        paddingTop="10"
        {...props}
      >
        {children}
      </VStack>
    </>
  );
```

Using the template

In the path "../pages" you will find a file named _app.tsx.

This file is were all the magic happens.

Every page created will be rendered based on the contents of this one.

```
import '../styles/globals.css'
import type { AppProps } from 'next/app'

function MyApp({ Component, pageProps }: AppProps) {
  return <Component {...pageProps} />
}

export default MyApp
```

For example if we access the /posts part of a url the page posts will be sent as a component inside this file.

The `<Component {...pageProps} />` part will be the contents of the posts page.

So now we now that our pages are represented by that Component Tag inside MyApp

What we need to do is make the Component a child of our template so every other page will load with the same template.

Since our Navbar uses ChakraUI components it is not enough to include the Template. In order for our other components to know what style we use for each one of the we need to include the <ChakraProvider>

The code will look like this:

```
import "../styles/globals.css";
import type { AppProps } from "next/app";
import Template from "../components/Template/template";
import { ChakraProvider } from "@chakra-ui/react";

function MyApp({ Component, pageProps }: AppProps) {
  return (
    <ChakraProvider>
      {/*ChakraProvider incapsulates the whole app*/}
      <Template>
        {/*Template incapsulates only the pages that will be rendered in the
future*/}
        <Component {...pageProps} />
      </Template>
    </ChakraProvider>
  );
}

export default MyApp;
```

Adding Authentication

In order to add authentication to our app we will use the next-auth library

Within the terminal we will run `yarn add next-auth` this will install the next-auth library required to let our users log in.

To make use of this library we need to include our app inside <SessionProvider> tags.

```
import "../styles/globals.css";
import type { AppProps } from "next/app";
import { ChakraProvider } from "@chakra-ui/react";
import Template from "../components/Template/template";
import { SessionProvider } from "next-auth/react";
```

```
// Add session to the current pageProps
function MyApp({ Component, pageProps: { session, ...pageProps } }: AppProps) {
  return (
    <ChakraProvider>
      {/*SessionProvider takes one attribute called session which we set it to
session, this will be automatically set upon user login*/}
      <SessionProvider session={session}>
        <Template>
          <Component {...pageProps} />
        </Template>
      </SessionProvider>
    </ChakraProvider>
  );
}

export default MyApp;
```

Now we can use sessions inside our application.

Now we need to add the signin and signout endpoints for our app.

Next-Auth API Endpoint

Inside our pages folder we will create the following path **"/api/auth/[…nextauth].ts"**

This is very important if the path is not the above mentioned the Next-Auth library will not know how to correctly signin and signout

The […nextauth.ts] is named "Catch all routes" every endpoint accessed under "/api/auth/" will access the logic inside this file.

Inside the file we need to call the NextAuth function from "next-auth library"

We will not use local providers in this tutorial.

We will make the app available to Sign In using a GitHub Account

In order to do that you will need to register an app with GirHub and get your clientId and clientSecret by accessing this link https://github.com/settings/applications/new

After getting the id and secret we will add them to the .env file inside the parent folder

```
GITHUB_ID = clientID
GITHUB_SECRET = clientSecret
```

[...nextauth].ts

```
import NextAuth from "next-auth";
import GitHubProvider from "next-auth/providers/github";

export default NextAuth({
  providers: [
    GitHubProvider({
      //@ts-ignore
      clientId: process.env.GITHUB_ID,
      //@ts-ignore
      clientSecret: process.env.GITHUB_SECRET,
    }),
  ],
  secret: process.env.NEXT_PUBLIC_SECRET,
  pages: {
    signIn: "/auth/signin",
  },
});
```

Auth endpoint explained

The NextAuth() function will take as parameter an object containing specific properties.

We will only use the providers, secret and pages properties for now.

- providers
  - This property is an array containing all registered providers
  - Here we will add our GithubProvider
  - The GithubProvider will require the clientId and Secret that you got from github and added inside the .env file
- secret
  - this is a public secret used to encode/decode the information received from the provider
  - you can generate one using a secure hash algorithm (for example `openssl rand -base64 32` will generate a random key)
- pages
  - this is an object containing the signin and signout pages if the default path is not respected.
  - We added this because we will create a separate signin page which will be opened in a modal after the user clicks sign in

Creating the Sign In Modal

First thing first, we need a custom SignIn Page that we can further render inside a modal.

Inside the pages folder we will create **"/auth/signin.tsx"** this will be our signin page. **Notice that it is the exact address that we offered the signin page inside the nextauth api endpoint.**

signin.tsx

The Sign In Page will be a NextPage that will accept providers as a parameter.

We will give this page all the providers that are defined inside the nextauth endpoint and for each provider we will generate a Sign In Button.

```
import { Box, Button, VStack } from "@chakra-ui/react";
import { NextPage } from "next";
import { BuiltInProviderType } from "next-auth/providers";
import { ClientSafeProvider, LiteralUnion, signIn } from "next-auth/react";

export interface SignInProps {
  providers: Record<
    LiteralUnion<BuiltInProviderType, string>,
    ClientSafeProvider
  >;
}

const SignIn: NextPage<SignInProps> = ({ providers }) => {
  return (
    <VStack>
      {Object.values(providers).map((provider) => (
        <Box key={provider.name}>
          <Button onClick={() => signIn(provider.id)}>
            Sign in with {provider.name}
          </Button>
        </Box>
      ))}
    </VStack>
  );
};

export default SignIn;
```

SignIn notions explained

Providers must be of type Record<LiteralUnion<BuiltInProviderType, string>, ClientSafeProvider>, this is because later we will use a function called getProviders() included in the next-auth library which has this return type.

Next we will take the providers and map through them generating a Button that onClick will call the signIn function with the id and with the name as button text of that specific provider.

```
{Object.values(providers).map((provider) => (
        <Box key={provider.name}>
          <Button onClick={() => signIn(provider.id)}>
            Sign in with {provider.name}
          </Button>
        </Box>
      ))}
```

Now we need a modal to load that Signin page into.

SignInModal.tsx

```
import {
  Modal,
  ModalOverlay,
  ModalContent,
  ModalHeader,
  ModalFooter,
  ModalBody,
  ModalCloseButton,
  Button,
} from "@chakra-ui/react";
import { NextPage } from "next";
import { BuiltInProviderType } from "next-auth/providers";
import {
  ClientSafeProvider,
  LiteralUnion,
} from "next-auth/react";
import SignIn from "../../pages/auth/signin";

export interface SignInProps {
  providers: Record<
    LiteralUnion<BuiltInProviderType, string>,
    ClientSafeProvider
  >;
}

const SignInModal: NextPage<SignInProps> = ({ providers }) => {
  const showSignIn = false;
  const handleClose = () => {};
  return (
    <>
      <Modal isOpen={showSignIn} onClose={handleClose}>
```

```
        <ModalOverlay />
        <ModalContent>
          <ModalHeader>SignIn with Provider</ModalHeader>
          <ModalCloseButton />
          <ModalBody>
            {/*SignIn page is called here as a component*/}
            <SignIn providers={providers} />
          </ModalBody>
          <ModalFooter>
            <Button colorScheme="blue" mr={3} onClick={handleClose}>
              Close
            </Button>
          </ModalFooter>
        </ModalContent>
      </Modal>
    </>
  );
};

export default SignInModal;
```

SignInModal notions explained

Just like before the SignInModal will need to take providers as a parameter so we can pass them further to the signIn page.

ChakraUi gives us an easy way to create modals by using the <Modal> component.

```
<Modal isOpen={showSignIn} onClose={handleClose}>
```

The modal components has 2 important attributes:

- isOpen – this tells the modal if it should be displayed or not by accepting a Boolean
- onClose – this is the function that will toggle isOpen to true or false (we will get to this just in a moment)

```
          <ModalBody>
            {/*SignIn page is called here as a component*/}
            <SignIn providers={providers} />
          </ModalBody>
```

Inside the ModalBody we will call the SignIn Page component in which we pass the providers

Opening the Modal by clicking on the SignIn Button from the Navbar

In order to open the modal from the SignIn Button on the Navbar we need a way to modify the isOpen variable from the Modal from inside the Navbar.

To do this we will use React's useContext hook.

React context allows us to pass and use data across our application without passing props from one component to another.

Creating the Context

In our parent folder we will create **"/context/globalcontext.ts"** this file will contain the data we need to open and close our modal.

globalcontext.ts

```
import React, { Dispatch, SetStateAction } from "react";

export interface SignInContext {
  showSignIn: boolean;
  setShowSignIn: Dispatch<SetStateAction<boolean>>;
}

export const ShowSignInContext = React.createContext<SignInContext>({
  showSignIn: false,
  setShowSignIn: () => {},
});
```

GlobalContext notions explained

Creating a React Context will allow us to create variables that we can provide to our entire app or specific parts.

The context only contains the variable names, types and of course can contain default values.

Since we are using TypesScript we need to specify the type of context we will provide.

We do this by creating an interface containing those variables

```
export interface SignInContext {
  showSignIn: boolean;
  setShowSignIn: Dispatch<SetStateAction<boolean>>;
}
```

Later in our code we will use a useState hook variables and attribute them to these.

```
const [state, setState] = useState()
```

The “**state”** variable in a useState can be any basic data type (number, Boolean, object), and of course it can be a custom type if necessary.

The **“setState”** variable in a useState hook will always be a Dispatch Function(A function that allows you to access and/or modify that state is a Dispatch), in our case this one will be a `Dispatch<SetStateAction<boolean>>`

Next step will be to create the context itself, we do that by using the React.createContext.

```
export const ShowSignInContext = React.createContext<SignInContext>({
  showSignIn: false,
  setShowSignIn: () => {},
});
```

Here we specify the default values of the state and useState variables. One will be false the other an empty function.

Adding the context to our App

In order to use the context we created we need to provide it to the app.

To do it we will go in our _app.tsx. here we will make use of the useState hook.

We will create a useState hook with the same variables that we defined in the context:

```
 const [showSignIn, setShowSignIn] = useState(false);
```

We will import the context we created:

```
import { ShowSignInContext } from "../context/globalcontext";
```

And last but not least we will provide our context just like some normal JSX tags

```
<ShowSignInContext.Provider value={{ showSignIn, setShowSignIn }}>
      /* Other components here     */
 </ShowSignInContext.Provider>
```

Everything contained inside the `ShowSignInContext.Provider` tag will have access to showSignIn and setShowSignIn

The _app.tsx file will look like this

```
import "../styles/globals.css";
import type { AppProps } from "next/app";
import { ChakraProvider } from "@chakra-ui/react";
import Template from "../components/Template/template";
import { SessionProvider } from "next-auth/react";
import { ShowSignInContext } from "../context/globalcontext";
import { useState } from "react";
```

```
function MyApp({ Component, pageProps: { session, ...pageProps } }: AppProps) {
  const [showSignIn, setShowSignIn] = useState(false);

  return (
    <ChakraProvider>
      <ShowSignInContext.Provider value={{ showSignIn, setShowSignIn }}>
        <SessionProvider session={session}>
          <Template>
            <Component {...pageProps} />
          </Template>
        </SessionProvider>
      </ShowSignInContext.Provider>
    </ChakraProvider>
  );
}

export default MyApp;
```

Using the created Context

To make use of the created context we have access to the useContext hook available from React.

useContext takes as a parameter the context we crated and returns the variables we specified.

```
const { showSignIn, setShowSignIn } = useContext(ShowSignInContext)
```

`{ showSignIn, setShowSignIn }` => this is just another way of object destructuring.

We can extract from the useContext Object only the variables that we need.

Say we do not need the setShowSigning function then we just extract the showSignIn variable so:

```
const { showSignIn } = useContext(ShowSignInContext);
```

Inside our SignInModal.tsx we will add the following.

```
import { useContext } from "react";
import { ShowSignInContext } from "../../context/globalcontext";
```

Instead of

```
  const showSignIn = false;
```

we will now use the useContext hook from above

```
  const { showSignIn, setShowSignIn } = useContext(ShowSignInContext);
```

And the handleClose function will use the setShowSignIn to modify the state variable showSignIn:

```
  const handleClose = () => {
    setShowSignIn(!showSignIn);
  };
```

The SigninModal.tsx file will look like this:

```
import {
  Modal,
  ModalOverlay,
  ModalContent,
  ModalHeader,
  ModalFooter,
  ModalBody,
  ModalCloseButton,
  Button,
} from "@chakra-ui/react";
import { NextPage } from "next";
import { BuiltInProviderType } from "next-auth/providers";
import { ClientSafeProvider, LiteralUnion } from "next-auth/react";

import { useContext } from "react";
import { ShowSignInContext } from "../../context/globalcontext";
import SignIn from "../../pages/auth/signin";

export interface SignInProps {
  providers: Record<
    LiteralUnion<BuiltInProviderType, string>,
    ClientSafeProvider
  >;
}

const SignInModal: NextPage<SignInProps> = ({ providers }) => {
  const { showSignIn, setShowSignIn } = useContext(ShowSignInContext);
  const handleClose = () => {
    setShowSignIn(!showSignIn);
  };
  return (
    <>
      <Modal isOpen={showSignIn} onClose={handleClose}>
        <ModalOverlay />
        <ModalContent>
          <ModalHeader>SignIn with Provider</ModalHeader>
          <ModalCloseButton />
```

```
          <ModalBody>
            {/*SignIn page is called here as a component*/}
            <SignIn providers={providers} />
          </ModalBody>
          <ModalFooter>
            <Button colorScheme="blue" mr={3} onClick={handleClose}>
              Close
            </Button>
          </ModalFooter>
        </ModalContent>
      </Modal>
    </>
  );
};

export default SignInModal;
```

Using SignInModal

Now all we have to do is use the SignInModal we created.

Inside the index.tsx file we will add the SignInModal as a component and provide it with it's necessary parameters

Index.tsx

```
import type { GetServerSideProps, NextPage } from "next";
import { BuiltInProviderType } from "next-auth/providers";
import {
  ClientSafeProvider,
  getProviders,
  LiteralUnion,
  useSession,
} from "next-auth/react";
import SignInModal from "../components/SignInModal/SignInModal";

export interface IndexProps {
  providers: Record<
    LiteralUnion<BuiltInProviderType, string>,
    ClientSafeProvider
  >;
}

const Home: NextPage<IndexProps> = ({ providers }) => {
  const { data: session } = useSession();
```

```
  return (
    <>
      <SignInModal providers={providers} />
      {!session ? (
        <> You must log in to see Posts! </>
      ) : (
        <> You are logged In! </>
      )}
    </>
  );
};

export const getServerSideProps: GetServerSideProps = async (context) => {
  const providers = await getProviders();
  return {
    props: { providers },
  };
};

export default Home;
```

Index notions explained

getServerSideProps – This is a nextJS specific function which if exported the page will be pre-rendered using the fetched data.

```
export const getServerSideProps: GetServerSideProps = async (context) =>
{
  return {
    props: {}, // will be passed to the page component as props
  }
}
```

Before the page is rendered the getServerSideProps will run and the data fetched will be used in the page.

More details here https://nextjs.org/docs/basic-features/data-fetching/get-server-side-props

In our case the function will fetch all the providers we defined to log In in our […next-auth] endpoint.

Then we pass them to our index page after which to our SignInModal.

```
  const { data: session } = useSession();
```

useSession returns the current session(this session will be available only if the user logs in using one of our providers.

Inside our index Body we can choose what to show our users based on the existence of the session

```
      {!session ? (
        <> You must log in to see Posts! </>
      ) : (
        <> You are logged In! </>
      )}
```

TO BE CONTINUED