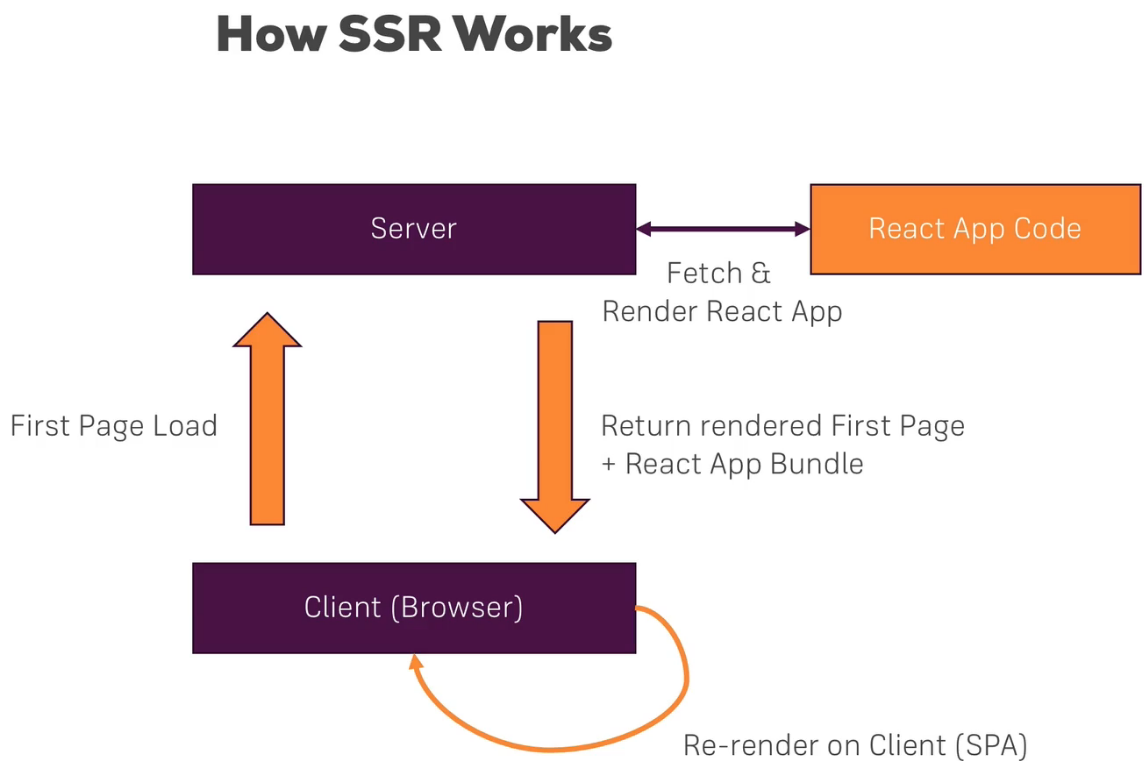
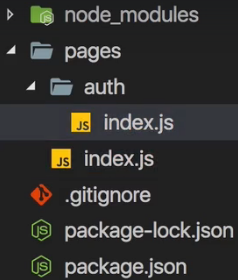
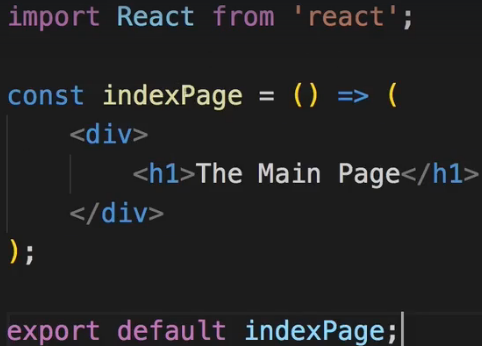
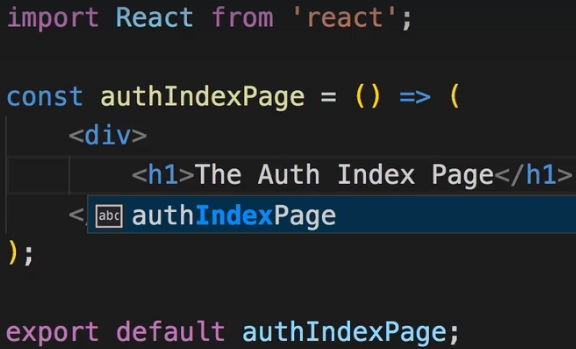
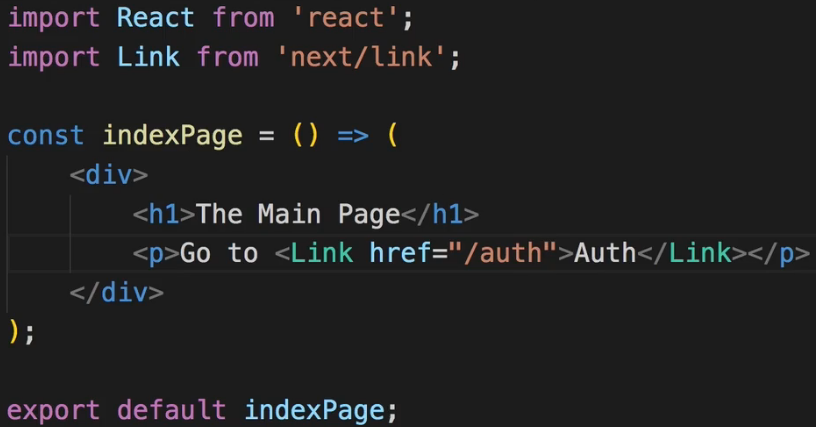
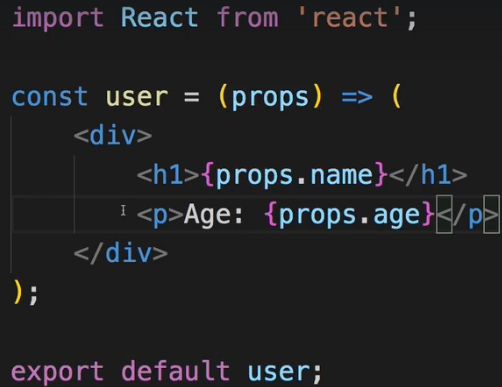
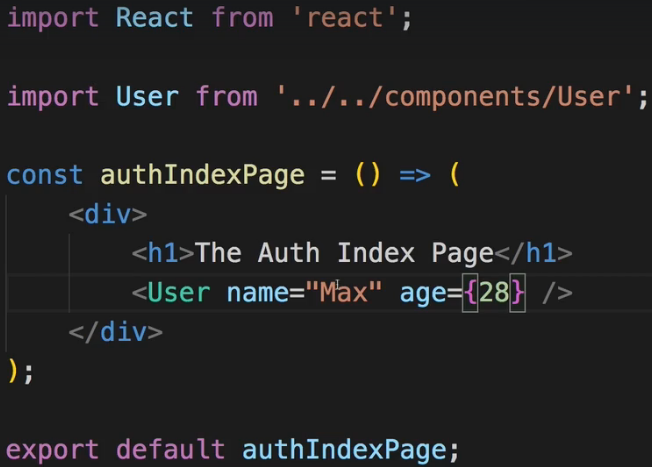
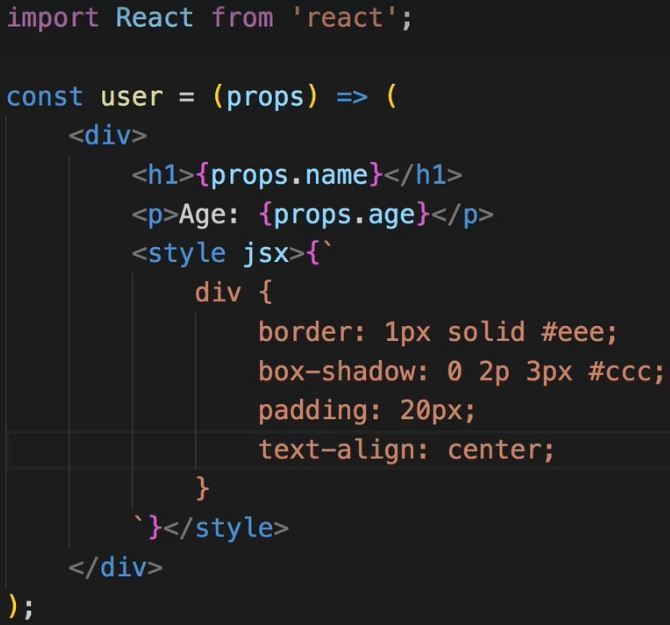
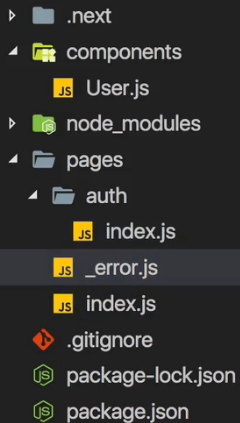
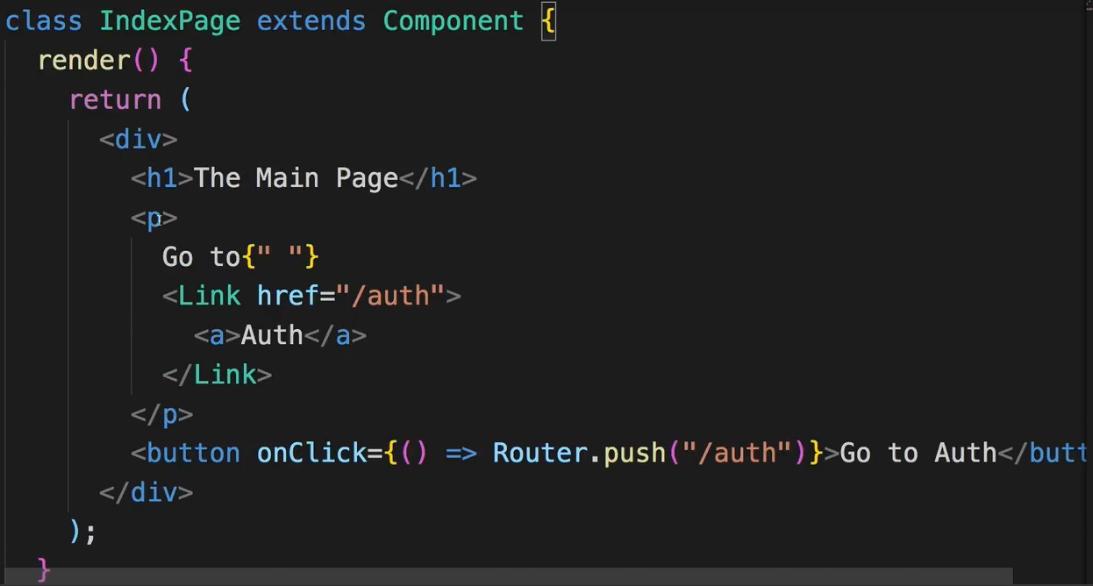
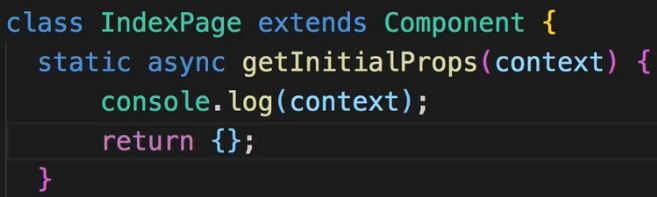
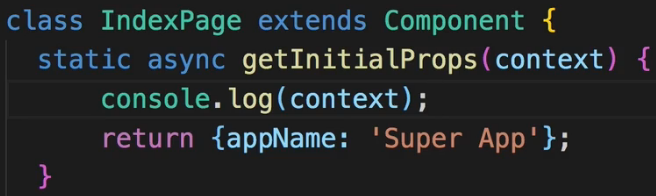
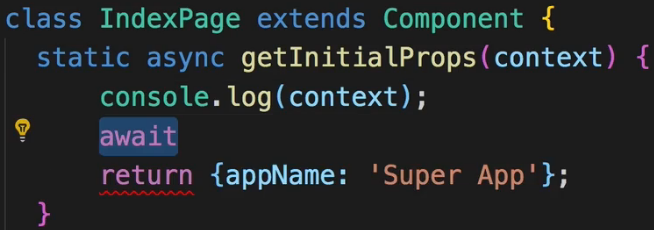
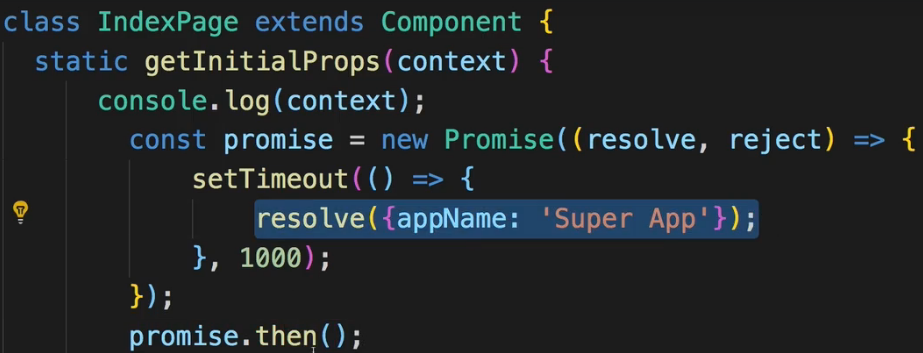
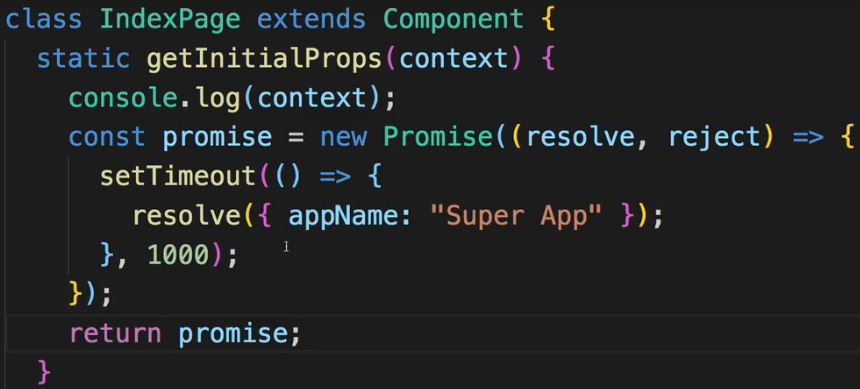
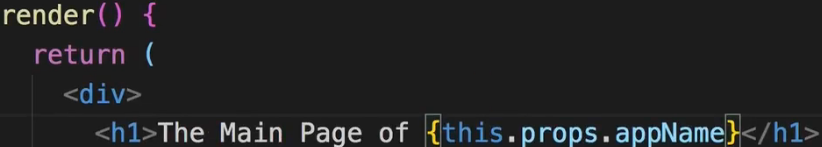
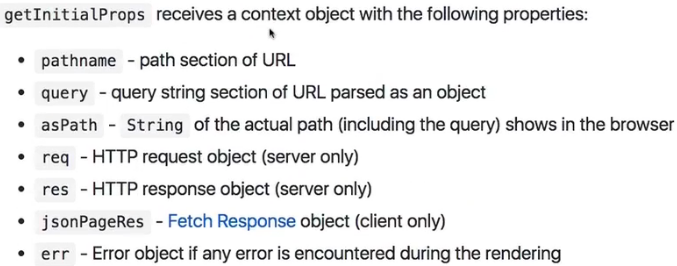
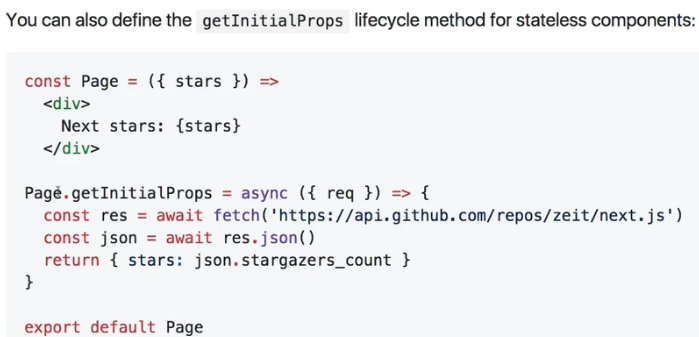
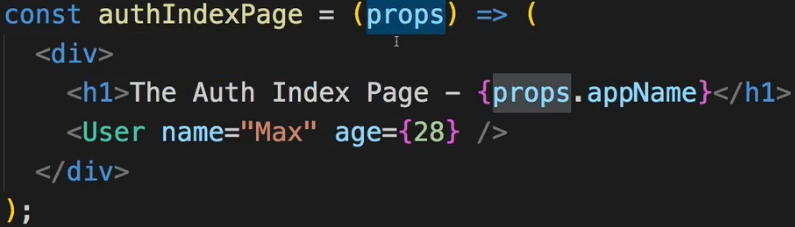
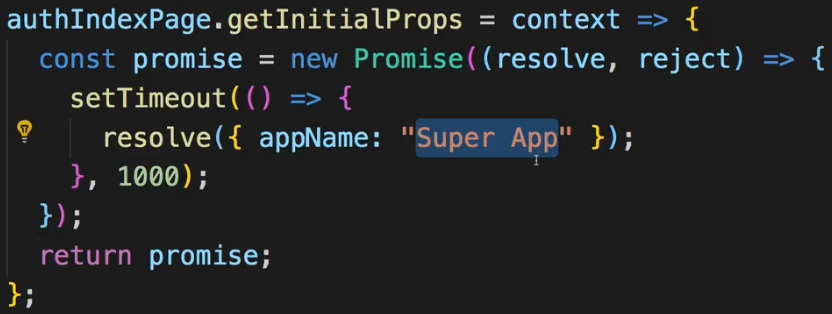
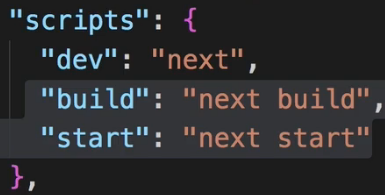
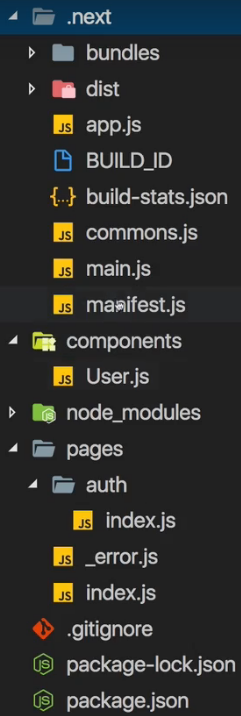
**Section 23 done: 10/10 Bonus: Next.js**  
**Introduction**  
\* NextJS: Beyond create-react-app.  
\* Next.js is a library building up on React and you’ll still be writing a React application with it, it’s not an extra library, framework or syntax.  
\* Next.js instead enforces a specific folder structure you have to use and gives you some things like server-side rendering out of the box, since you use that folder structure, it can manage a lot of things for you so that you don’t have to configure them manually and therefore it can make the creation of React apps more convenient. Especially the server-side rendering part.  
\* Server-side rendering is a super interesting part of React which you might not need but which can be helpful if you have an application that’s highly dependent on being found in search engines for example.  
\* We’ll cover the important core concepts in this module.  
**Understanding Server Side Rendering**  
\* Next.js is a minimalistic framework for server-rendered React applications.  
\* Server-side rendering is quite an advanced React concept. There are therefore courses focusing entirely on that.  
\* In a nutshell server-side rendering simply means that when a user enters a URL leading to your React app or 1 of your pages of your React app, it will pre-render that page on the server so that the server returns that pre-rendered HTML code.  
\* Normally when you visit the server for the first time so you load the page for the first time, the server would typically ship you the index.html file and your code and then you’d load it as a SPA in your client (browser).  
=> The problem can be that if your page needs to be crawlable by search engines, crawlers like the Google Crawler might not really understand or read your page correctly, especially if you’re using Asynchronous code to load the initial content. So if you have a page where the first thing you do in JavaScript is reaching out to a server and then render the results to the screen when the data is back, that might happen super fast to the user, but the Crawler will only see the spinner or whatever you’re showing until the data is there.  
\* Now with server-side rendering for the first page load, which is always what the Crawler sees, the server will fetch and render the React app. So it won’t just fetch and return it, it also renders it, for that page you visited and then it returns that pre-rendered page along with the React app bundle, so that from this point on, you still have a SPA so you then still work as before, it’s just about that initial page load essentially.  
  
\* Now Next.js is a package helping you with server-side rendering because that is super tricky to set up. And besides just helping you with it, it gives you a specific structure to work in. This structure takes advantage of enforcing a strict folder structure you have to use - though we’re only talking about 1 folder mainly which has to be named in a certain way - so that all the Routes of your application are actually generated automatically.  
**Setting Up a Project**  
\* You can find all the instructions on the GitHub page of Next.js.  
\* To run these commands you need an empty project.  
\* I just have .gitignore file.  
**npm init**  
**npm install --save next react react-dom**  
\* We should now add this “scripts” to package.json.  
  
\* With that we can start using it and now we have that important file system thing I was talking about.  
\* In Next.js the file system is our main API. We don’t use React Router with Next.js, instead, we create folders and files to reflect our URLs in the file system. And Next.js - the package - will automatically parse that and use its own internal Router to handle all the heavy-lifting.  
\* Now along the way it pre-renders the content we load as pages, on the server, it automatically code-splits so Lazy Loads that, all of that out of the box without us configuring anything. That is why Next.js is great, a lot of stuff is working out of the box there at the price of having to keep a certain file structure.  
\* That file structure is mainly about tha `pages` folder.  
=> Let’s creating a `**pages**` folder. And in there you now create your JavaScript files which will be React components, which will hold and export React components, and these filenames now define different paths you have.  
\* You can also have a sub-folder like `auth` there and in there add for example user.js. You would reach this file at yourDomain/auth/user. If you name it index.js, you’d reach it at yourDomain/auth.  
\* Let’s also add an index.js file inside the `pages` folder, now we have something like example.com/ and would reach that file.  
\* Now let’s add a component to both, and typically here you use Functional components but of course you can also use Stateful ones with the class keywords in case you need to manage state.  
    
\* If we run `npm run dev`, this will spin up a development server, building your application and server-side rendering it, also giving you stuff like Hot Reloading so a really nice development environment.  
**npm run dev**  
\* If you now go back to the browser, you can actually load this in a new tab, under localhost:3000.  
\* There you should see your page and if you now go to /auth, you see the `auth` index.js page.  
\* So without us setting up any Routing, we can load these 2 different components as pages.  
**Understanding the Basics**  
\* I just want to show you the bare basic features so that you can dive deeper if you want to.  
\* Let’s now see how we can easily navigate from our main index.js page to the `auth` index.js page.  
\* For that we of course need a link and with React Router we used the Link component.  
\* Now with Next.js we also have a Link component we can use, it’s not coming from React Router.  
  
\* We can point to /auth to point to the `auth` folder and there it’ll always pick the index.js file if you got one.  
\* Link alone won’t do the trick though, you actually still use the normal <a> anchor tag inside of it, without the href attribute though. Behind the scenes Next.js will basically add href attribute to the <a> tag but then capture any clicks on the <a> tag and handle that internally, so that you don’t really reload the page.  
\* Another cool thing you should be able to see is: if you go to the Network tab, and reload your main page, and you click on auth, you also see that an extra auth bundle was loaded, and that is what Next.js does for you - automatic code-splitting (Lazy Loading).  
\* By the way don’t be intimidated by the file sizes, this isn’t optimized at all, this contains a lot of development environment overhead.  
\* So that’s the automatic Code-Splitting / Lazy Loading you get for free.  
\* And with the Link you see here, you can easily navigate around in your application.  
\* You’re also not limited to using the Link though - you can for example navigate around using a button like this:  
  
\* And this is a very quick way of imperatively - so in code - navigating around.  
\* This is how you get around, how you use the link in Next.js, one of the core concepts, and it looks super simple but a lot is happening behind the scenes, most importantly the automatic code-splitting, which is awesome.  
**Next.js & Components & Pages**  
\* So we learned the basics about Next.js.  
\* But what if I still want to compose my application from components and not just pages. Should I not do that anymore?  
=> You absolutely should. The way you build your React application doesn’t change, it’s still a good practice to compose your application of many small pieces.  
\* Let’s add a `components` folder and you can name it whatever you want, Next.js doesn’t care.  
\* Let’s add a User.js component in there.  
   
\* Once we have it, we use it as always. Let’s say we want to use it in the `auth` index.js file.  
\* So we still work with components in our Next.js application.  
\* We just have this special `pages` folder, which also contains components - Function or Class-based, but which is the only folder which gets some special treatment because Next.js will parse all the files in there as pages, create Routes for them, and automatically code-split.  
\* By the way, the code-splitting of course also includes components included in that page, like the User component. We only use that in the `auth` index.js file so we only load the code for it if we navigate to that page.  
**Styling our App in Next.js**  
\* In create-react-app we used CSS Modules but I taught you that you have other options too.  
\* Next.js has its own way of styling React, you can still use inline styles, you can still use Radium. You can’t use CSS Modules because you can’t access the Webpack configuration Next.js uses.  
\* But it offers another out of the box working way of styling your components and having scoped styling.  
=> You add a **<style>** HTML element to your component JSX. And then on that style you set `**jsx**` as a prop. The `jsx` is a prop understood by style here because Next.js uses a special package which is called **styled-jsx** - you can learn more about it on the GitHub page.  
\* And styled-jsx is a package which just allows you to write your CSS styles here in your JSX code and then it’ll pull them out, apply them to your component here and scope it to that component. Here you can also use media queries and so on.  
\* Important: you need to wrap all of that in **{``}** which allows you to create string literals so basically multi-line strings you could say.  
  
\* So we use this third-party package styled-jsx to create styles which are scoped that component.  
**Handling (404) Errors**  
\* On the GitHub page of Next.js we can scroll to Custom error handling.  
\* Because right now we already have Routes now in our Next.js application but if I enter an invalid route here, we get this 404 error. Better than nothing but we can customize this.  
\* You can create your own error handler by creating a **\_error.js** file in the `pages` folder.  
   
\* So this is how we can handle errors in Next.js and how we can render our own custom error page.  
**A Special Lifecycle Hook**  
\* Let me show you 1 super important feature of Next.js.  
\* Let’s turn our index.js in `pages` folder into a Class-based component.  
  
\* I want to show you a special Lifecycle Hook you can use.  
\* **And unlike the React Lifecycle Hooks, that Lifecycle Hook would even be available in Functional components**.  
\* I’ll show you how to use it there too but let’s start in the Class-based one.  
\* It’s called **getInitialProps()**.  
=> It’ll receive an ARGUMENT which describes the `**context**` of your application.  
\* Now to see what’s inside of that, we need to change it because it’s not a normal Lifecycle method like the other ones, instead it’s a `**static**` method which means it can be called without the component being instantiated yet and it’s an `**async**` Asynchronous method which means it doesn’t resolve, it doesn’t return something instantly.  
\* Let’s make sure we return an object. It can be empty. Because it should resolve to an object.  
  
\* We can now see it in the console where we run `npm run dev`. Because this code actually is executed on the server first, it’ll only be executed on the child if you navigated there within the app so by clicking a link for example. If you navigated there by typing the URL and hitting enter or refresh, it executes on the server, that’s the special thing about this Lifecycle method. It executes either on the server or the client and you can use it to initialize your app before it loads. So on the server for example in getInitialProps() you could fetch data from a database and then pre-populate the props this component will receive, with props of your choice.  
\* For example here we could hard-code an appName prop, which of course here isn’t fetched from some database but you could easily do that.   
  
\* Since it’s an `async` method, you can use the `await` keyword in here to await for a PROMISE to resolve and you would typically have a PROMISE or something like that if you are reaching out to a web server for example with Axios.  
  
\* Now if you don’t want to use `async` here, what you can also do is use a normal static method like this (in case you don’t know that async await syntax) and then you’d create a PROMISE or use a package like Axios which does something that returns a PROMISE. In a PROMISE you typically execute Async code and again oftentimes you don’t write PROMISES on your own but you use some package which wraps some Async action like reaching out to a server in a PROMISE, Axios does that for example.  
  
\* Now this .then() part is what we leave out here, instead we simply return the entire promise in our getInitialProps() method. And Next.js will take care about listening to its result and then pre-populate our props once the result is there and only render the page once the result is there.  
  
  
\* And we know that this will be populated because we pre-populate the props here in our getInitialProps() function.  
\* Now we have to wait that 1 second and then it displays the “Super App” because it did pre-populate our props. So getInitialProps() - super important, working together with PROMISES, either those written by you or returned by third-party packages, it’ll wait for them to resolve and pre-populate your props.  
\* Now let’s take a look at this `context`.  
\* Now it’s quite hard to read that here in the console because we see all the nested properties and so on too.  
=> So let’s instead have a look at the official documentation of Next.js.  
\* There if we go to the getInitialProps() part - which is the: Fetching data and component lifecycle.  
  
\* Important: on the server only though, the req and res objects. So basically the same you’d get in an Express.js environment. This allows you to really get all the information of the Requests that reach the page. You might not need that but there are also some use cases where this can be really helpful as it allows you to analyze what exactly the user requested and so on.  
\* So here in index.js we’re not taking advantage of the incoming `context` data, we could do so though of course, we’re nonetheless using this to initialize our props which is an important function of this method. It’s probably the reason why it’s called getInitialProps().  
\* Now how do we use this in a Functional component?  
=> This is possible, you can see how it works on the official documentation too.  
  
=> You simply create your const and then add this as it is a `static` method, to that const after you created it. A function, which the Functional component of course is in the end, allows you to add something to it inside of that function. A function which this const here in the end is, is just a JavaScript object so we can add `static` methods to it.  
  
\* Then if we click on the auth link, after 1 second we’re taken there and see it.  
\* So getInitialProps() - super useful Lifecycle which works in both Functional and Class-based components and which allows you to initialize your props before the page is actually loaded.  
**Deploying our App**  
\* You can always visit the official GitHub page for Next.js.  
\* And you can visit learnnextjs.com, the official page by the creators of Next.js which requires a GitHub account to log in but which then step by step walks you through the core features.  
\* You can now build a React app which is server-side rendered by default, where you don’t need to worry about code-splitting / Lazy Loading and Routing and where you therefore might have an easier time creating the React app you’re looking to create.  
\* Let me show you how to prepare this for deployment.  
  
=> `build` is the script you want to execute to build the project for deployment.  
**npm run build**  
\* It’ll now compile and optimize all that code and spit it out, ready for you to ship to a server.  
\* Once that is done, you get the `.next` folder. This now contains all the build content, now this `dist` folder here is what actually needs to be deployed but you would deploy the entire project folder because you also need to install node\_modules and so on. So you would deploy the entire project folder here to a HOST like AWS S3 or Heroku and that’s important: you need a HOST, a SERVICE which is able to run Node.js. Because Next.js uses Node.js and only works on Node.js. All the files you deploy are JavaScript files, so you need a Node.js-ready HOST and then you ship your entire project there and just execute `**npm start**`, the 3rd script.  
  
\* We can also run `npm start` locally since we got Node.js installed here, this will now still serve the app in localhost:3000 but now it’s the app we prepared for deployment, it works as before but it’s actually using the optimized files where everything is minified and so on and it shows that everything works.  
\* And therefore if you ship the entire project, and then execute `**npm install**` followed by `**npm start**` on your server, you have your React Next.js application running on a real server, with all the benefits Next.js gives you.  
**Next.js Repo & Docs**: <https://github.com/zeit/next.js/>