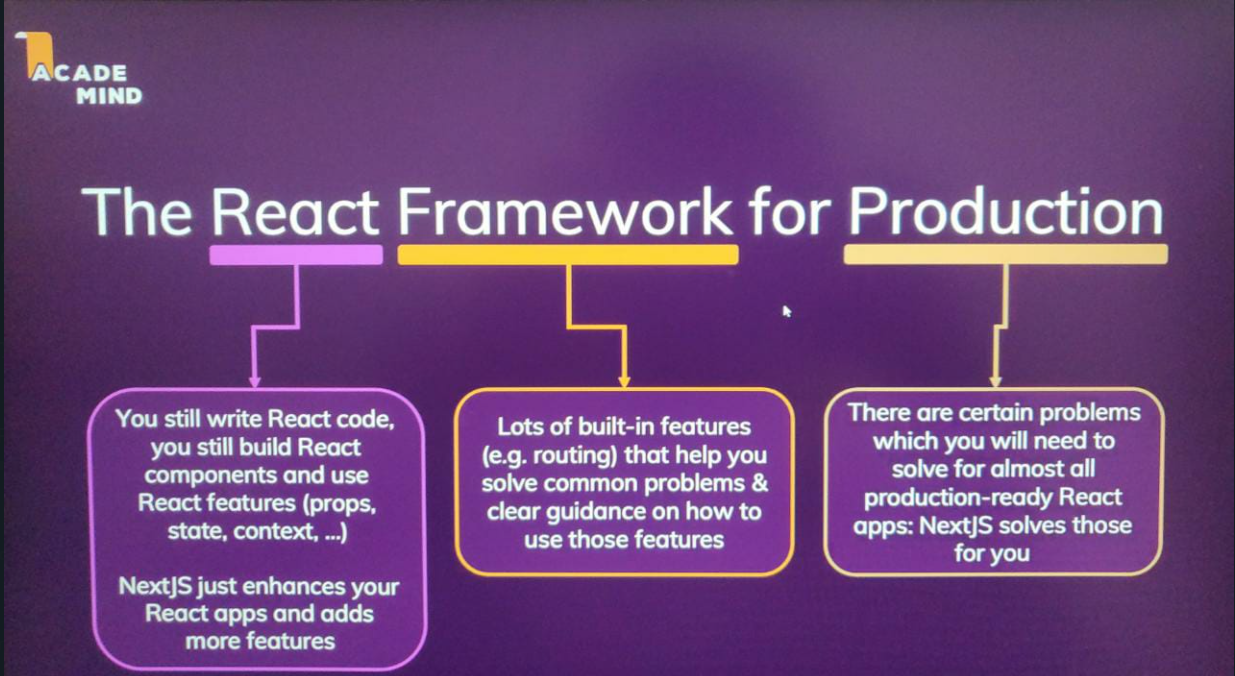
**React** is a Javascript Library for building User Interface.

**NextJS** is the React framework for production. It helps making large scale react apps easier.

Now the difference between a framework and a library in the end is that a framework is bigger. It has more features than a library. It's focusing on more things instead of just a single thing. And it's also giving you clear rules, a clear guidance on how you should write your code, how you should structure your files and so on.



**Key Features**

1) **Server-Side Rendering**: with react we perform client-side rendering, i.e., the rendering is done by react and being a client side javascript library all this rendering is done on the client side browser and not the server. As a result the actual HTML code sent from the server to the client is pretty empty (because thereafter it is rendered on the client side).

It depends on what you're building, but it can be a problem. Because for example, if your page also, fetches some data from a server that should be displayed like a list of meetups, as we're doing it here, then the user might initially see some loading state, a flickering page for a fraction of a second, whilst the request is on its way, fetching the data because data fetching only begins once the JavaScript code executed on the client. And then we still have to wait for the response of that outgoing request. Simply because the page which we requested did not yet contain that data. Now, again, that is not necessarily a problem, but of course it might not be the user experience you want to offer.

Another problem can be SEO. So, for example, here, where we got this list of meetups, we see those meetups as a user but the search engine crawlers will actually only see that initially empty HTML page which we're getting from a server. So, that content is not picked up by search engine crawlers.

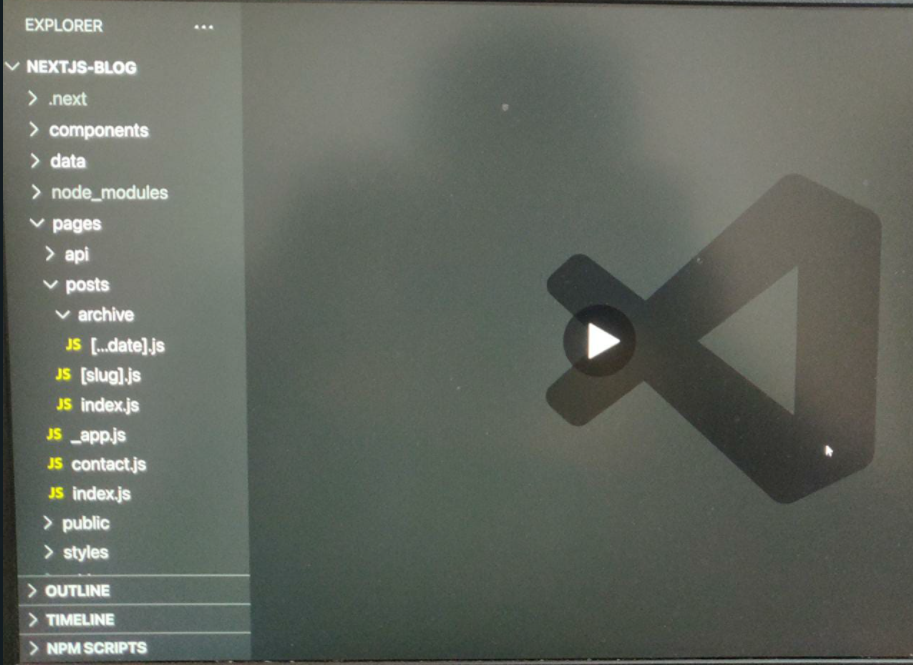
And that's where a server-side rendering could help us. If that page would be pre-rendered on the server, if that data fetching somehow could be done on the server, when the request hits that server and then the finished page would be served to our users and to the search engine crawlers, then users would not have that flickering loading state and search engines would see our page content. And that's the problem server-side rendering solves. It allows us to pre-render React pages.

Now ReactJS actually has built-in features that allow you to add server-side rendering but it can be tricky to get that right. And it requires extra setup from your side. With NextJS, that becomes way easier because NextJS has built-in server-side rendering. It automatically pre renders your pages and that means that with NextJS, if you build a standard NextJS app, without any extra setup from your side, if you visit such a page, it was pre-rendered on the server by default out of the box. And that means that it's great for a search engine optimization because search engines see what our users see and our users also have a better initial load experience because they don't have that initial flickering. If we inspect the source code of a NextJS page, it's half a page in the NextJS app, then we see that there, it's not just an empty HTML skeleton, but instead all the content is already there in that HTML page, which we got back from the server. Now it is worth noting that with NextJS, after this initial load offered as initial request, we still get a standard React app running in the browser.

2) **File Based Routing**: In traditional React, you don't even have a router. And just to be clear, routing means that we give the user the illusion of having multiple pages. When we navigate around and we load different pages, then that's the job of a router. Typically, we use React Router for that.

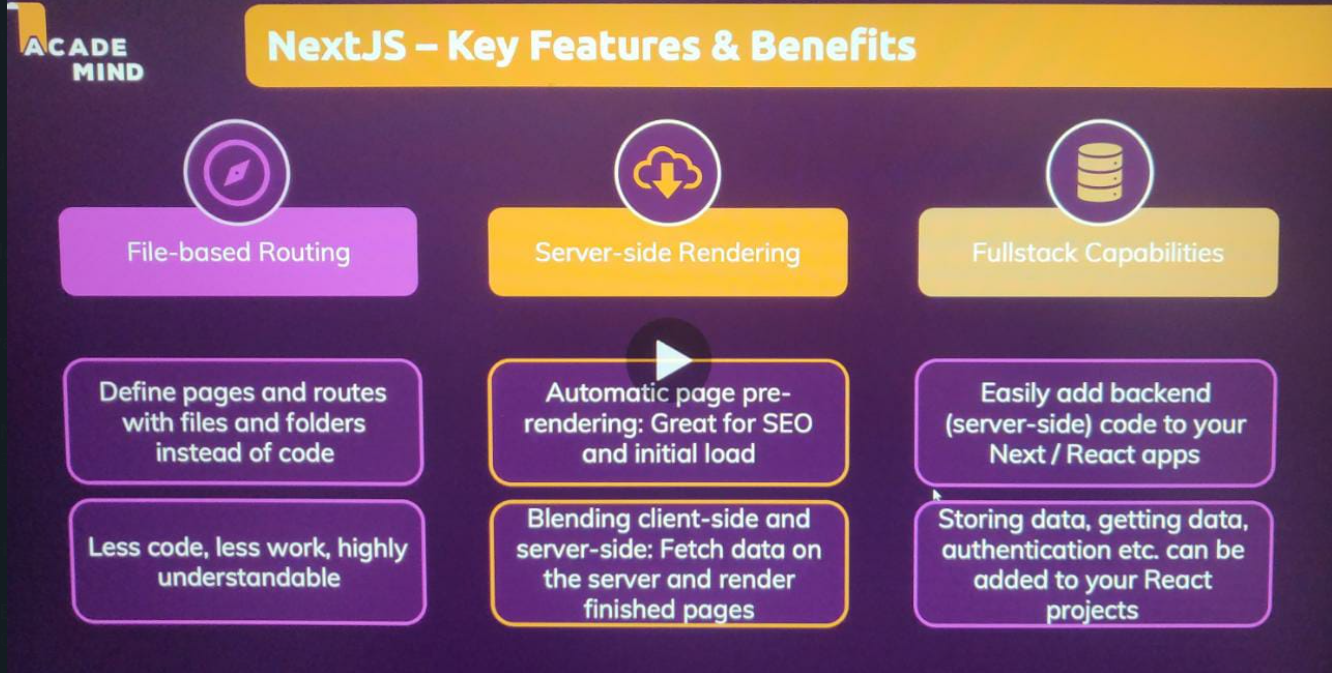
This router basically watches the URL, and when it changes, it basically prevents the browser default of sending a request to some backend server, and instead renders different content on the page with React. A different component in the end. That's what routing is. We change what's visible on the screen based on the URL without sending an extra request to a server because we stay in that single page application which we typically build with React.

NextJS gets rid of that in-code route definition (done in react). Instead, with NextJS, you would define pages and routes with files and folders. You have a special pages folder in NextJS apps which has to be named pages, and then you are structuring that folder, defines the routes and paths your page supports.



Hence, we have to write less code.

3) **Build Fullstack Apps**: Add backend code easily. NextJS also makes it easy for us as a developer to add backend code to our react project. So to build a full stack react project, where we don't just have the client side code maybe with service side pre rendering, but where we also have standalone backend code, that for example, works with the file system or reaches out to a database. With NextJS it's very easy to add our own backend API into our react project using NodeJS code. So we can easily add such code to our Next react apps when using NextJS. And that allows us to add code for storing data to a database or to files, getting data from there, adding authentication and all off that.



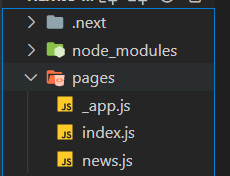
**Getting Started**

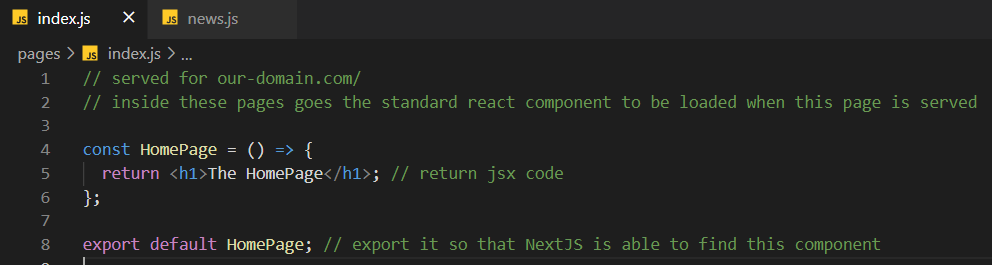
**Here in the NextJS app, we don’t have the index.html file. And the reason for this is that NextJS has this built in pre-rendering. And whilst it gives you a single page application, that single page is dynamically pre-rendered when a request reaches the server so that you do return an initial page with content.**

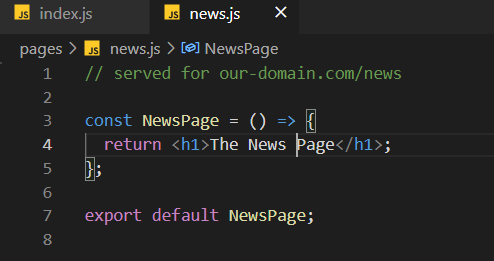
The pages folder will be the most important folder because that is where we will set up that file based routing, and that is there for the folder, which is important for us to define the different pages that should make up our application here.

**SIDENOTE: Within the pages folder, the index.js file will be our route page. So if a request reaches our domain slash nothing just index.js will be loaded.**

The file name will be used as a path name (except for index.js which is served when we have a request to our-domain). For example the file name **news.js** is served for a request to **our-domain.com/news**.







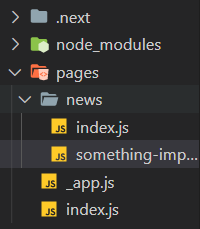
With these 2 basic code, if we visit <http://localhost:3000> we get the index page and by visiting <http://localhost:3000/news> we get the news page.

Also, 1 important thing to notice is if we view the page source, then we see that in there we don't just have an empty skeleton, but if we look up at closer, we find the actual page content in here. And that's an important difference to a standard React app where the page is not pre-rendered on the server. This HTML code which we see here is the HTML code sent back by the server. And since it contains that content here that means that the entire page was pre-rendered. And that's why the content ends up in here and the advantage's that we don't have any flickering on the page whilst we're waiting for it. And in addition, search engines would also see that content here.

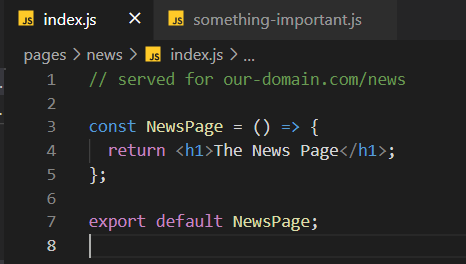
Hence, we did file based routing and used server side rendering without any external method.

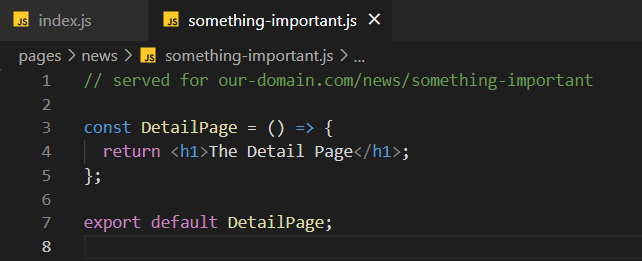
**Nested Routes**

**Also, the folders which we create within our pages folder act as path segments. This means if I add a news folder and within it add a file named index.js, then it is also loaded for** <http://localhost:3000/news>. This is important if we want to have 2 segments, i.e., like <http://localhost:3000/news/something-important>, where something-important is acting as an identifier (because if we create files directly in the pages folder we are limited to 1 segment). Hence, to have nested path with more than 1 segment we create a different folder (for which we can also make a root page by naming the first file within it as index.js).

****

**The below is the index.js file within the news folder which acts as the root page for this folder routes.**

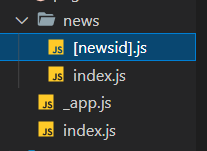
****

****

**Dynamic Pages with Parameters**

If the index.js file within news has many links, then it is not good to hardcode the identifier as **something-important** in the file name. We want to create a dynamic page, where the path segment can be dynamic (eg <http://localhost:3000/news/something-important2>), so that we have multiple identifier but we always load the same page no matter what the concrete value here is. But then inside of the page, we simply have access to that value in the path so that we can fetch the proper data.

If you have square brackets in your file name like this, this tells nextJS that this will be a dynamic page so that it should be loaded for different values in your path. And then you can add an identifier between those square brackets where the identifier name is totally up to you. Something like newsId, for example, like this.



**[newsid]** has the same code as **something-important** on last page.

This then tells nextJS that this page will be loaded for different values. So for example, for something important after slash news but also for any other identifier.

So that is how we can add such a dynamic path here. And that is another key feature of nextJS. It's a feature that allows us to build truly dynamic and flexible websites with nextJS.

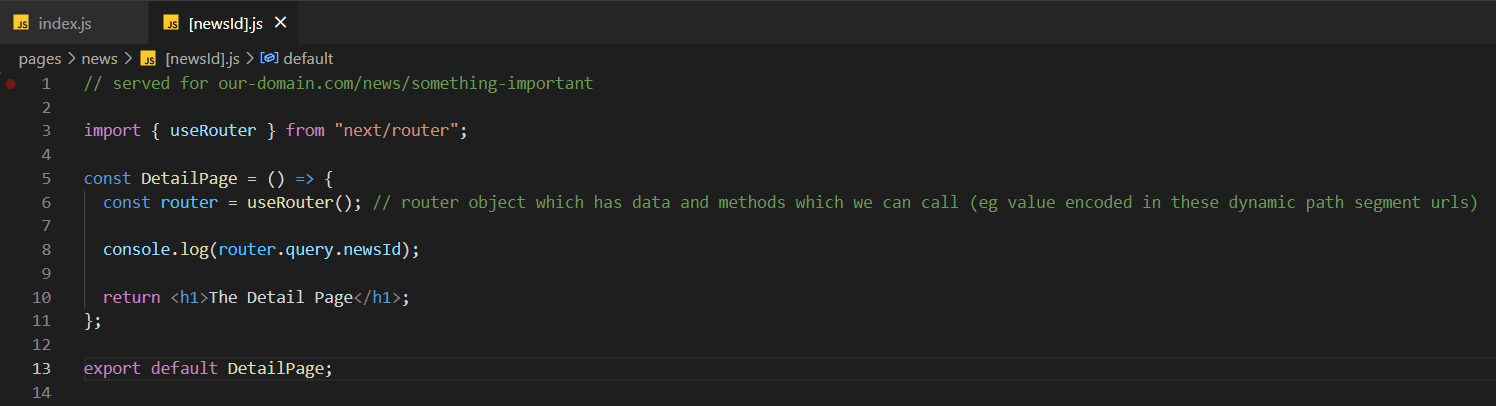
But how can we now extract the entered path value inside of the component so that we can, for example, then fetch the correct news item from a database, let's say when a user visits this page?

**Extracting Dynamic Parameter Values**

To extract the concrete value entered in the URL when this page is loaded Next.js gives us a special hook which we can use. A special react hook which we can use in functional components.

**useRouter** hook is a custom hook built by the react team.

And getting access is easy on this router object we've got this query property which gives us access to a nested object and on that query object we then have the identifier which we chose between the square brackets as a property name. So, in my case newsId because that's what I entered here as a file name between the square brackets. And that will then hold the concrete value in the URL for this dynamic segment for which this page was visited.



Hence, we get access to the concrete data encoded in the url. We can now use this to get our id and if we have a backend api where we can use fetch, we can send a request to get the news with this specific newsid.

And that's how we can build dynamic pages which work for different pieces of data and then can do different things based on different pieces of data. Here, we could fetch different news items from a database based on the different Ids for which we visit this page.

**Linking Between Pages**

Currently we change the url to visit multiple pages which we don’t want. Hence, if we use **<a href = >** we can achieve linking. But this is only the second best way of navigating around in a NextJS app. It works but watch this refresh icon here when I click this link. It briefly turns to a cross and then goes back to the refresh icon. This always signals that the browser sends a new request and gets back a new HTML page. And that all works but it has a disadvantage. It means that we don't have a single page application here. It means instead that we're always sending a new request to the backend to fetch a new HTML page whenever the user navigates around. And that's not our goal here.

It's great that we have those pre-rendered pages so that when a user initially visits our site, some content is there right from the start and it's also great for search engines but if a user is on the page already and then navigates around on it, we wanna stay in that single page application because that allows us to preserve state across pages and give the user a better, a more reactive experience, a better user experience. That, after all, is one of the reasons for using React, that we wanna build an interactive UI where we never send a request for a new HTML page but where we instead update what's on the screen with JavaScript, with React in the end.

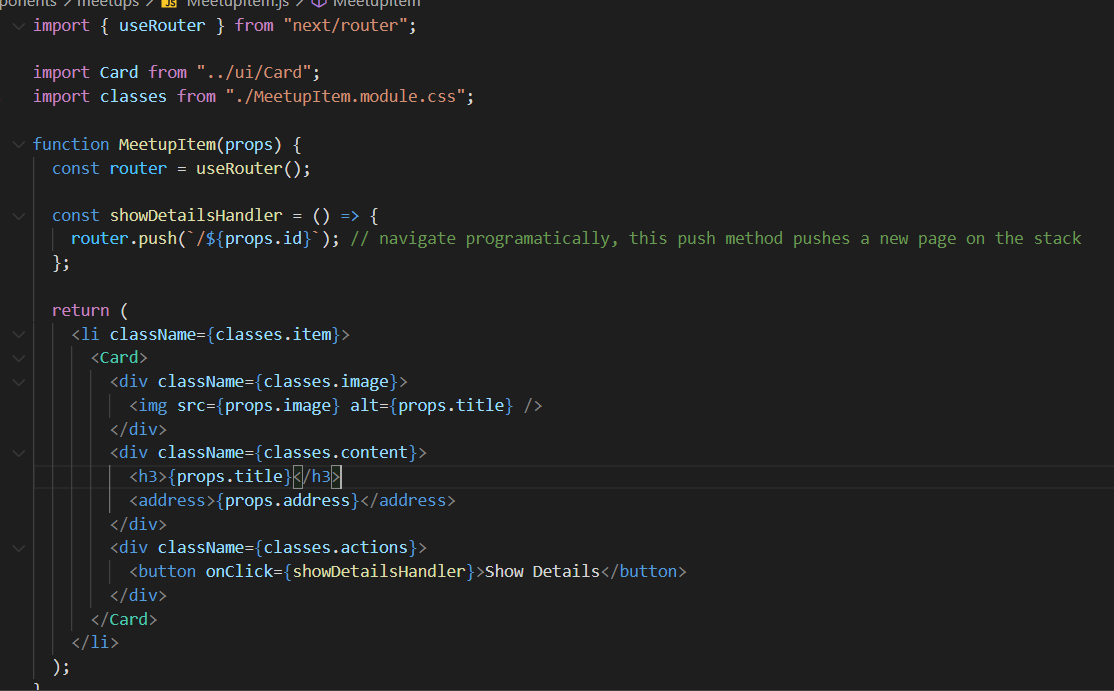
And we're not doing this here when we create a link like this. Instead, we load a brand new page. We would lose any state we might be storing, any Redux sate or context state, all of that would be lost because we load a new page. And we simply don't have a single page application here.

By using the following **Link** component we can achieve the best results.

Now we instantly go to the second page and now we go there without fetching a new HTML page. Instead we are on a single page application and the visible content on the screen is actually just re-rendered by React. And that's great because that allows us to combine the best of both worlds. We have this highly interactive and reactive single page application here where we can manage and store state across pages

Because Link, this special Link component renders an anchor tag but it watches clicks on those anchor tags and if you click there, it prevents the browser default of sending a request, at getting a new HTML page. Instead, it will load the to be loaded component for you and change the URL so that it looks like you changed the page whilst in reality, you stay in that single page application.

**Navigate Programmatically using useRouter Hook**

****

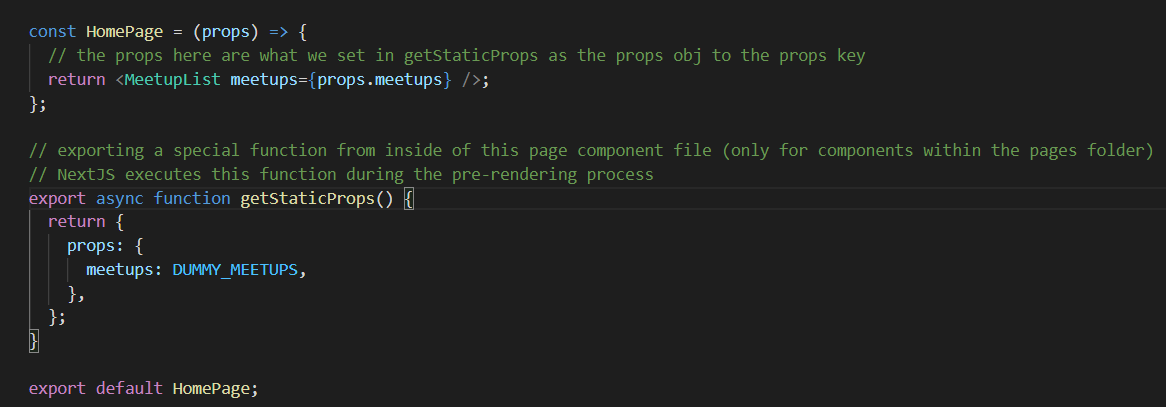
**.module.css allows to scope classes to a component. (so that class names are unique per component and cant spill to other components) Classes is the object and all the css classes we define will be available as properties on this javascript object.**

**Watch video 332 to understand how we need to fetch data from server in such a way that nextJS pre-renders it and the search engine does not get the empty list, but the list after fetching is done.**

Exporting a special function from inside of this page component file (only for components within the pages folder)

NextJS executes this function during the pre-rendering process

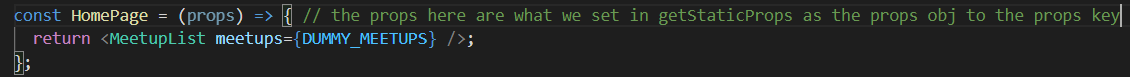
So it will then not directly call your component function and use the returned JSX snapshot as HTML content but it will, first of all, call getStaticProps before it calls the component function. And getStaticProps has this name because indeed, its job is to prepare props for this page. And these props could then contain the data this page needs. And that's useful because getStaticProps is allowed to be asynchronous. You can return a promise there and then, and that's the key thing, NextJS will wait for this promise to resolve, which means it waits until your data is loaded and then you return the props for this component function. And with that, you're able to load data before this component function is executed so that this component can be rendered with the required data.



Now, here in getStaticProps, you can also execute any code that would normally only run on a server. You could access a file system here or securely connect to a database because any code you write in here will never end up on the client side and it will never execute on the client side simply because this code is executed during the build process, not on the server and especially not on the clients of your visitors. So the code in here will never reach the machines of your visitors. It will never execute on their machines.

But then once you're done with whatever you did to get the data you need, you need to return an object here in getStaticProps. You always need to return an object here. Now, in this object, you can configure various things but most importantly, you typically set a props property here and it has to be named props. And that then holds another object, which will be the props object you receive in your component function here in this page component function. This now receives a props object and the object will be the object you set as props here in getStaticProps. And there we could have our meetups key in there. The structure of this props object is totally up to you, which holds our DUMMY\_MEETUPS. With that, those DUMMY\_MEETUPS would be loaded and prepared in getStaticProps and then they would be set as props for this page component. Therefore, in this page component, we no longer need to manage state, we no longer need useEffect and we can therefore get rid of those imports here because now we get the data through props. And our meetups for the MeetupList component are props.meetsups. .meetups because I'm adding a meetups prop down there. And that's how we can move the data fetching away from the client (because earlier first pre-render was done and then react took over and the second rendering in which the data fetching was done by use-effect was done on the client side) to the server-side or to be precise to the during the build process side.

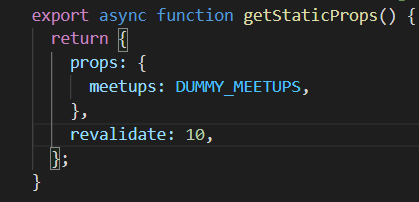
If I now view the page source, we see that we no longer have an empty unordered list, instead we have an unordered list, which has list items with the images and the title and so on. So now this is pre-rendered and it now contains the full HTML code and that's, of course, also great for search engines then because now, data is not fetched in a second component render cycle on the client but initially, before this page is pre-rendered, during the build process.



**Re-validate Property**

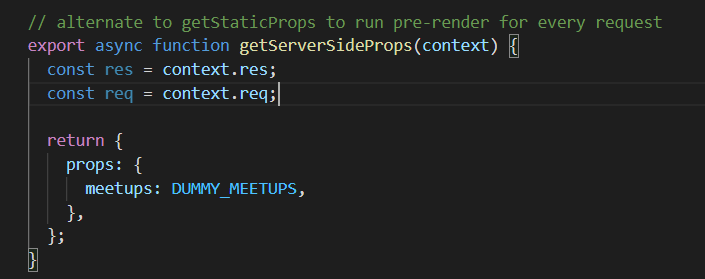
And one pretty big problem, which we could face in some websites, depending on what we're doing is that the data here could be outdated. This page, as I mentioned, is generated during the build process. So thereafter, we deploy it. If we then add more meetups to our database, this pre-generated page would not know about them. And if we don't add any client-side data fetching, we would always just see the outdated meetups here. And this could, of course, be a problem. Now, we can always rebuild our site and redeploy when our data changes. And for some websites, like personal blogs, this is a great alternative because there data doesn't change too frequently but if data does change more frequently, there is a extra property, which we can add to this returned object. And that's the revalidate property. When we add this property to the object returned by getStaticProps, we unlock a feature called incremental Static Generation. Revalidate wants a number, let's say 10, and this number is the number of seconds NextJS will wait until it regenerates this page for an incoming request. That means that with revalidate set to some number, this page will not just be generated during the build process. It will be generated there but not just but it will also be generated every couple of seconds on the server, at least if there are requests for this page. So that means that this page, with revalidate set to 10, would be regenerated on the server at least every 10 seconds if there are requests coming in for this page. And then these regenerated pages would replace the old pre-generated pages. And with that, you would ensure that your data is never older than 10 seconds.

But whatever you set this number to, you will ensure that this page will occasionally be re pre-generated on the server after deployment so that you don't have to redeploy and rebuild all the time just because some data changed.



**Pre-rendering the Page for every request**

Now with revalidate, you can ensure that this page is also updated regularly after deployment. But sometimes even a regular update is not enough. Sometimes you really want to regenerate this page for every incoming request. So, you want to pre-generate the page dynamically on the fly after deployment on the server. Not during the build process and not every couple of seconds, but for every request. And if that's your goal, then there is an alternative to getStaticProps.



And the difference to getStaticProps is that this function will now not run during the build process, but instead always on the server after deployment. (Any code you write in here will always run on the server, never in the client. So you can run the server side code in here, you can also perform operations that use credentials that should not be exposed to your users, because this code only runs on the server)

Now you can't set revalidate here, because it doesn't make any sense here. This getServerSideProps function runs for every incoming requests anyways, so there is no need to revalidate every x seconds.

**That works exactly as we learned it, but now their page is really pre-generated for every incoming request.**

It's guaranteed to run for every request. But that actually can be a disadvantage, because that means that you need to wait for your page to be generated on every incoming request. Now if you don't have data that changes all the time, and with that, I really mean that it changes multiple times every second. And if you don't need access to the request object, let's say for authentication, getStaticProps is actually better. Because there you pre-generate an HTML file, that file can then be stored and served by a CDN. And that simply is faster than regenerating and fetching that data for every incoming request. So your page will be faster when working with getStaticProps, because then it can be cached and reused, instead of regenerated all the time. Hence, you should really only use getServerSideProps if you need access to that concrete request object, because you don't have access to request and response in getStaticProps. Or if you really have data that changes multiple times every second, then therefore even revalidate won't help you.

**getStaticPaths**

Because since this is dynamic, NextJS needs to know for which ID values it should pre-generate the page. Because how would it pre-generate this page otherwise? We get the ID from the URL here. Great, but keep in mind that this is not pre-generated when a user visits this page with a specific value in the URL, but during the build process. So here we need to pre-generated for all the URLs, for all the meetup ID values users might be entering at runtime. And if they enter an ID for which we didn't pre-generate the page, they will see a 404 error. But because that is how it works, we need to add getStaticPaths which has the job of returning an object where we describe all the dynamic segment values.

And getStaticPaths is a function you need to export in a page component file. If it's a dynamic page like we have it here and you're using getStaticProps, not if you're using getServerSideProps and not if you're using neither get static nor getServerSideProps but it is needed if you do use getStaticProps.

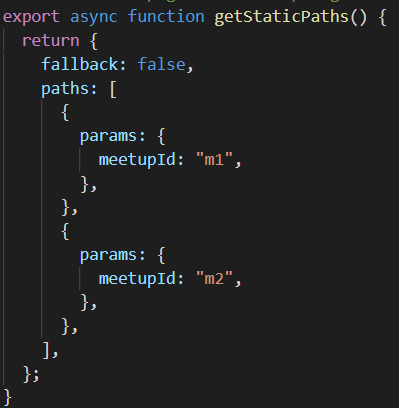
It has the job of returning an object where we describe all the dynamic segment values. So all the meetup IDs in this case, for which this page should be pre-generated.



Now for this, this object needs to have a paths key, which is an array. And in that array, you must have multiple objects one object per version of this dynamic page. Where this object has a params key. That's a must have, which then itself again is an object with all the key value pairs that might lead to your dynamic page. So if you have multiple dynamic segments, then you would have multiple keys in this nested object. Here we only have meetup ID as a single dynamic segment. And hence here in this params object, we would add a meetup ID key and then enter the concrete value for meetup ID for which this page should be pre-generated. And if we have multiple possible values. Like in this case where I have, M1 and M2 we would return a paths array with two objects inside of it, where the other one uses M2 as a meetup ID.

Now, in reality, you would of course not hard-code this as a developer, but you would also fetch your supported IDs from a database or from an API and generate this array dynamically. And we are also going to do that later, but for the moment let's hard-code it.

And there’s another piece of configuration, which you need to add in this returned object next to your paths key, the fallback key. This key tells NextJS whether your paths array contains all supported parameter values or just some of them. If you set fall back to false, you say that your paths contains all supported meetup ID values. That means that if the user enters anything that's not supported here, for example, M3 he or she would see a 404 error. If you set fall back to true on the other hand, NextJS would try to generate a page for this meetup ID dynamically on the server for the incoming request.



Fall back is a nice feature because it allows you to pre-generate some of your pages for specific meetup ID values. For example the pages which are visited most frequently and then pre-generate the missing ones dynamically when requests for them are coming in.

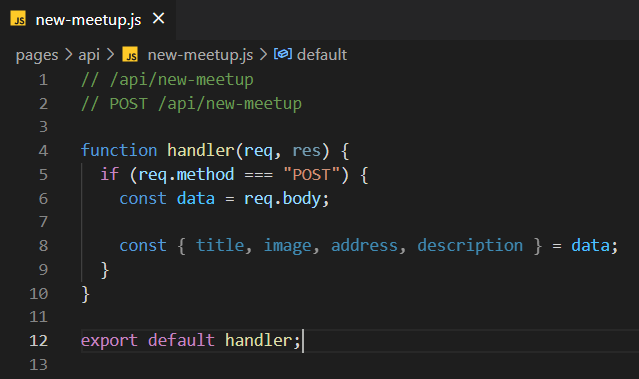
And getStaticPaths therefore is another important function, which you need in dynamic pages to tell NextJS for which dynamic parameter values this page should be pre-generated. And then again, getStaticProps executes for every page. So for every meetup ID value allows you to fetch data for that meetup and allows you to return props for that meetup.

**API Routes**

Then the NextJS will pick up any JavaScript files stored in there (**in api folder**) and turn those files into API routes. So into end points, that can be targeted by requests and that should receive JSON and return JSON. Now in this API folder, you can then again add JavaScript files where the file names will act as path segments in the URL.

Now, in those JavaScript files here, you then don't create a React component function. These API routes are not about defining, rendering or returning React components. Instead in there, we will define functions which contains server-side code because API routes will only run on the server never on the client. Decoding them will never be exposed to the client. So we can also use credentials in API routes without compromising them. And those functions are then simply triggered whenever a request is sent to this route, so to /api/new-meetup here. This would be the URL of this file and if a request is sent to this URL, it will trigger the function which we have to define in this file.

This function will receive a request and a response object. You might notice from node.js and express.js. The request object contains data about the incoming request. The response object will be needed for sending back a response.

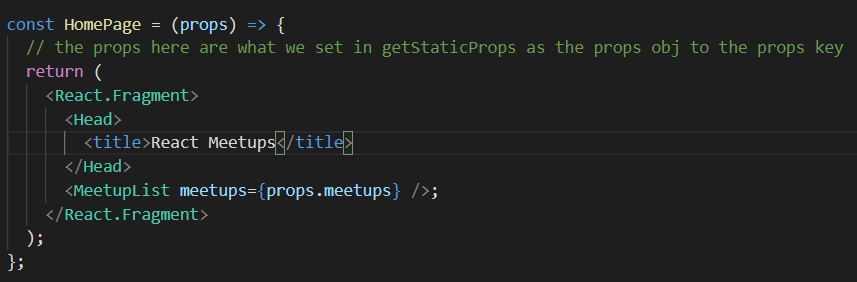


**Adding head MetaData**

Import the **head** component from next/head.

This is a component which allows you to add Head elements to the Head section of your page. You simply add it to your returned JSX code.

And now inside of these Head tags here between the opening and closing Head component tag, there you can now insert your Head elements. So all the HTML elements, which you can add in the Head section, you can add them here. Like for example, the title.



And now with that, we really prepared our page for deployment, our website for deployment because with that, we make sure that search engines get all that extra metadata and we also show our users some nice titles here as well.