**Next.js (**[**https://nextjs.org/**](https://nextjs.org/)**), Astro.js (**[**https://astro.build/**](https://astro.build/)**)**

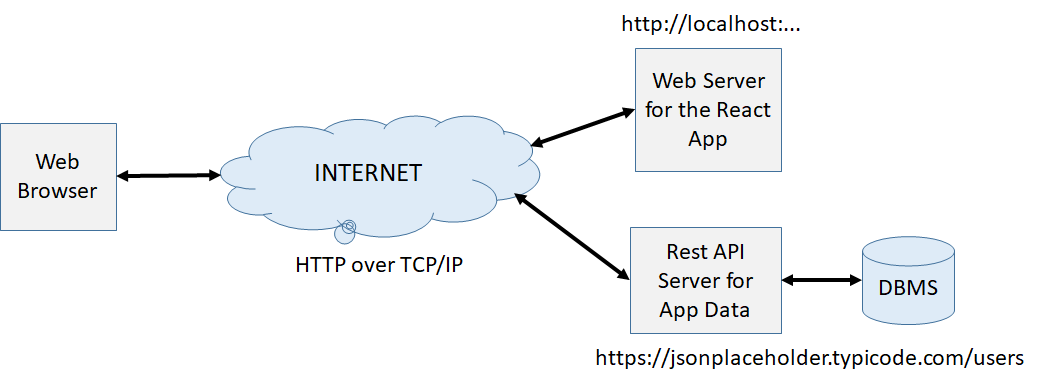
In this course our focus was on front-end JS frameworks utilizing the **Single Page Application (SPA)** architecture, in which all routes in a page (including the root page) are rendered at the browser using JS. This is called **Client Side Rendering (CSR)**. Although this paradigm is good for highly interactive Web applications, it has the following problems:

1. **The first page load is usually slow**: Since the root page in an SPA is empty and needs to be dynamically generated at the browser by JS, it usually takes much longer for the page to appear on the browser compared to a **Multi Page Application (MPA)**, in which static HTML pages are served from the Web server. This means that the performance metric *Time-to-Interactive* (TTI) (<https://web.dev/interactive/>, <https://pagespeed.web.dev/>) for SPAs is usually not good. Furthermore, some edge devices (desktop, laptop, tablet, smart phone, etc.) may be really slow (slow CPU/not enough memory etc.), which makes the page load even slower causing frustrated users. Lastly, since the JS code to generate ALL pages of the application is bundled into a single JS file, which is sent to the client when the root page is loaded, the size of this JS bundle is usually very big. There are ways to lazy load the JS code for different pages and components, but this requires extra work. For a tutorial on how to do lazy loading look here: <https://www.youtube.com/watch?v=JU6sl_yyZqs>
2. **Search Engine Optimization**: The root page in an SPA comes empty with a single root div element. This means that Web crawlers downloading the page will see no content at all until the page is rendered at the client. This causes a lot problems for Search Engines in that they do not have any information as to how to index this page. But if the page has a lot of content, the Search Engines can utilize the data in the page to correctly index the page so that it can easily be found during Web searches.

In order to observe these problems closely, consider a React app that has 2 static pages, and 2 dynamic pages as follows (15-NextJs-Astro/react-user-list):

1. /, which is the landing page of the app with links to other pages and gives information about the app
2. /about, which gives information about the company/university etc.
3. /users, which gives a list of users working at the company (current users are dynamically fetched from a Rest API endpoint)
4. /users/id (e.g., /users/1, /users/2), where id is a number such as 1, 2, .., which gives detailed information about the user with the given id. The details of the user is fetched from a Rest Api endpoint.

To implement this application we used React for the front-end (We could in fact use any of the front-end frameworks from Angular, Vue, Svelte, Qwik, …). But notice that we also need to implement a backend Rest Api to serve the list of users and the details of the user given their id. In this course we emulated this backend using json-server or talked to already existing fake Rest API endpoints, but in reality we have to implement the Rest API using some sort of a backend PL framework (Node.js+Express, PHP+Laravel, Python+Django/Python+Flash/Python+FastAPI (for a list of Python Web frameworks, please refer to <https://wiki.python.org/moin/WebFrameworks>, <https://hackr.io/blog/python-frameworks>, <https://www.netsolutions.com/insights/top-10-python-frameworks-for-web-development-in-2019/>), Java Spring Boot, ASP Core, Ruby on Rails, …) and a DB (MongoDB, MySQL, MS-SQL, …). Here is how our 3-tier architecture will look like:



As you can see, we have a Web server that serves our React app (<http://localhost>:..) and a Rest API server (<https://jsonplaceholder.typicode.com/users>) that serves our app data, the users. The Web browser first contacts the Web Server to download our React app along with its JS bundle. To see the JS bundle created for our app, run “% npm build”. This will create a “dist” directory where you can find “index.html” and the JS bundle for the app. You can now serve this distribution by the “serve” utility (<https://www.npmjs.com/package/serve>), which is used to serve a static site, single page application or just a static file. After installation, simply run:

|  |
| --- |
| % serve dist –p 8000 |

This will create a Web server running at port 8000. You can contact this Web server as <http://localhost:8000>. Open the network tab and look at the files being downloaded from the Web server.

We know that the React app HTML page comes up with an empty root div element, so the browser must run the JS code to render the root page (**/**). After the root page is rendered, we can see a navigation bar at the top with links to other pages (routes). When we go to the **/about** page, the browser directly renders this page using **CSR** (by running the About React component). Notice that this page only contains some static data, which means that the browser does not have to contact any external resource to render this page. To see that the browser does not contact any external resource to render the **/about** page, open the Network tab in your browser debug window and see that there is no network activity when you switch between the **/** and **/about** pages.

When we go to the **/users** page, our react app uses the fetch API of the browser to contact our Rest API server (<https://jsonplaceholder.typicode.com/users>) to get a list of users and then renders them inside the browser. This is handled by the **Users.jsx** page in our app. We can now see that the browser sends a GET request to the Rest API server to fetch this data (which is usually small), and then renders this data inside the DOM using **CSR**.

When we go to a dynamic route such as **/users/1** to get the details of a user, then **UserDetails.jsx** page in our app is rendered, which first contacts the Rest API server to fetch the details of that particular user (with id=1) and then renders it inside the DOM. Again a GET request is sent to the server and the user data is received as a JSON object.

In order to show that this is a SPA and the global app state is maintained throughout the lifetime of the app, we also implemented a simple CounterCountext and wrapped the App component with this context so that all components can make use of it. In our app, the About component makes use of this context and display the current value of the counter and two buttons with which you can decrement or increment the counter’s value. Observe that as you switch between the pages, this global state is never lost. Therefore, when we come back to the /about page, we continue seeing the current (the last) value of the counter. Had this been a Multi-Page Application (MPA), after we switch from one page to the next, a new page would have been loaded from the server and the current value of the counter would have been lost. You can see this by simply re-loading the app and observing that the counter value goes back to 0 after a reload.

Looking at this application, we can clearly observe the problems associated with a SPA that we mentioned in the introduction:

1. The JS bundle sent for our React app contains logic to generate all 4 pages of our app: /, /about, /users, /users/id, which bloats the size of the JS bundle. In reality, however, we will only see the root page (/) when the page first loads. The other pages will be rendered only when requested, which may never be the case! This large JS file download and parse increases *Time-to-Interactive*.
2. It would be great if we could separate the HTML/JS required for each page into separate files and only serve the HTML+JS bundle for the page when the page is actually asked for from the server. This will reduce the size of the initial JS tremendously and will only send the HTML+JS for each page separately when asked for. Because the size of the initial JS bundle is now minimized, this will make our app more responsive by reducing *Time-to-Interactive.*
3. Notice that both the / page and the /about page contain only static data. This means that we don’t really need JS to render these pages. It would be great if we can generate the HTML for these static pages offline (called **Static Site Generation (SSG)**) and simply serve these HTML pages on demand!
4. Because the root page (/) now contains only static data and can be generated statically and served separately as an HTML file, the SEO is achieved. The search engines can download our static pages, which actually contain a lot of HTML elements, and index them properly, which was NOT possible with React.
5. We implemented the front-end of our app using React, but we still need another backend PL framework to implement the Rest API endpoints for our app. That is, we need API endpoints for /users and /users/id to serve the list of all users and the details of a user given their id.
6. Sometimes we might want to generate the HTML for a page at the server on demand when the request arrives (called **Server Side Rendering (SSR)**) rather than at the browser using **CSR**. In fact, a lot of the backend frameworks work on this model: They are based on the MVC architecture and render all pages at the server using a template engine.

The question is the following: Can we design a JS framework based on React that solves all of the problems mentioned above? Enter **Next.js**!

**Next.js (**<https://nextjs.org/>**)**

Net Ninja Next Tutorial: <https://www.youtube.com/playlist?list=PL4cUxeGkcC9g9gP2onazU5-2M-AzA8eBw>

Traversy Next Crash Course: <https://www.youtube.com/watch?v=mTz0GXj8NN0>

Traversy Static Blog with Next.js and Markdown: <https://www.youtube.com/watch?v=MrjeefD8sac>

Next.js is a framework that lets you implement the front-end and the backend of your project in a single project structure. The front-end is based on an extension of React.js and still uses the SPA model. But it allows you write pages that can be rendered at the client (CSR), statically at the server (SSG), dynamically at the server (SSR). It also permits incremental static site regeneration and has TypeScript support, smart bundling (a separate HTML+JS bundle for each page), route pre-fetching, and more. The backend APIs are implemented in a syntax very similar to Express and must be served inside a Node.js container. You can therefore deploy your Next.js app to any hosting provider that supports Node.js. Here is a list of providers for Next.js:

* AWS Copilot
* Digital Ocean App Platform
* Google Cloud Run
* Heroku
* Railway
* Render

In the following, we will be taking our User List React app and convert it into several Next.js apps with different features.

To create a next.js app with some boilerplate code, type:

|  |
| --- |
| % npx create-next-app@latest <name of your app> |

We first created a Next.js app that generates two static pages (/, /about) using **SSG**, and two pages that are rendered at the client (/users, /users/id) using **CSR** (15-NextJs-Astro/next-user-list1). This is basically the same React app but implemented using Next.js. Notice that for each page in our app we have a .js file under /pages directory:

1. \_app.js wraps around all pages and corresponds to our App.js in a React app. This is where we included our CounterContext and provided it to all pages and components.
2. index.js is responsible for rendering the root page (/)
3. about.js is responsible for rendering the /about page
4. users/index.js is responsible for rendering the /users page
5. users/[id].js is a dynamic route responsible for rendering /users/id pages, e.g., users/1, /users/2, etc.

To start running this app in development mode, run

|  |
| --- |
| % npm run dev |

This will create a live server to start serving your app. Any changes you make to the app will automatically be detected and re-rendered.

Open the networks tab in Chrome debug tools and see the network activity as you walk over different pages. You will see that when you load the app, only the necessary html+js bundle responsible for the rendering of the root page will be downloaded. When you go to a different page, the html+js bundle responsible for the rendering of that page will then then be loaded. This is one of the benefits of writing your app in Next.js that we mentioned about: The HTML+JS code necessary for the rendering of a page is downloaded the first time you go that page. Therefore, the bundles are quite small as opposed to a pure React app, which has to bundle the JS necessary to render ALL pages into a single JS file.

In order to see how Next.js generates pages for production, run:

|  |
| --- |
| % npm run build |

This will create a .next directory, which contains all statically generated HTML files for all pages and all JS files necessary to render each page put into separate files. You can start serving this built by running:

|  |
| --- |
| % npm run start |

This will launch a live server that serves the files in .next directory. We can also export the build into an out directory so that it can be served using a regular Web Server as follows:

|  |
| --- |
| % npx next export |

This will create a directory named “out”, which you can start serving by the serve utility as follows:

|  |
| --- |
| % serve out –p 8000 |

So far we have seen the benefits of using Next.js compared to React.js: The base HTML for ALL pages are statically rendered during build time. The JS necessary for CSR is then bundled into separate files for each page and sent to the client when the page is first visited. As you can see, / and /about pages are entirely statically rendered at the server. We can see their HTML. Obviously, there must be some JS code that must be sent with these pages to handle button clicks.

As for pages /users and /users/id, the HTML for these pages are pretty much empty. We know that these pages are rendered using CSR by UserList and UserDetails components: Each of these components use the fetch API to fetch some data from a Rest API server and then generates the HTML inside the browser.

**API routes (Implementing a Rest API)**

Our current Next.js app has two routes /users and /users/id, which make use of two components UserList and UserDetails, both of which use the fetch API to fetch some data from a Rest API and then use this data to render themselves. Recall that both of these components talk to a fake Rest API called <https://jsonplaceholder.typicode.com/>. It is usually the case that it is our responsibility to implement these Rest API endpoints. Next.js lets us do this by putting the routes for these APIs under /pages/api directory. Specifically, we will implement two Rest API endpoints in our app: pages/app/users and pages/app/users/[id]. We also change UserList and UserDetails components so that they talk to our endpoints instead of the fake rest api. Here is the code for this app: (15-NextJs-Astro/next-user-list2).

**Server Side Rendering (SSR)**

Next.js allows us to render the pages at the server upon each request. This is called Server Sider Rendering (SSR), and is employed by many backend frameworks. In order to make use of SSR in Next.js, we have changed our previous app so that /users and /users/id pages, which were being rendered at the client using CSR, is now being rendered at the server using SSR: (15-NextJs-Astro/next-user-list3). When you go through this example you will notice that the two components (UserList and UserDetails) are now completely removed from our app. Instead, all rendering occurs inside the page components /pages/users/index.js and /pages/users/[id].js. When you look at these two files you will notice that both have a function named “getServerSideProps” that gets called first to generate the props to be passed the component that will render the page. In this example, we make a call to the fake rest API (jsonplaceholder.typicode.com) to fetch the users inside **getServerSideProps**, put them into a props structure and return it. This props will then be passed to the component at the top that renders the page. Notice that this function gets called every time we go /users page and generates the entire HTML for the page at the server. So, there is no CSR for these pages.

It would usually be the case that we will go to our database to fetch the users data instead of going to a fake rest api. 15-NextJs-Astro/next-user-list4 is a re-implementation of the previous app so that we get the user data from a local users.json file instead of fetching the data from a remote fake API.

One important thing to notice: When you implement Rest API endpoints or Server Side Rendering in your app, you can NOT export it into static files using “npx next export”. You can only export to static HTML files only if all your pages are statically generated (SSG) or have CSR components.

**Static Site Generation (SSG)**

Sometimes it is the case that you have some data that almost never changes such as a list of recipes, a list of blog posts, etc., and you would like to use this data and generate static Web pages. Since the data is pretty much static, we can use the Next.js **Static Site Generation (SSG)** property. To illustrate this, we have assumed that the user data that we have been using so far is static. That is, the list of current users are pretty much static and does not change. We know that our pages /, /about are already statically generated. Now, we will also statically generate pages /users /users/[id]. But the data for these two pages must come from a remote resource such as a database, a Content Management System, or another Rest API. In the following example, we will be fetching this data from our fake Rest API and then statically generate these pages.

Look at 15-NextJs-Astro/next-user-list5

When you look at /pages/users/index.js, you will notice that we now make use the function **getStaticProps**, which is used to prepare and pass props to the component that renders the /users page. When you look at /pages/users/[id].js file, you will see that we have another function called **getStaticPaths**, which gets called first to generate a list of “ids” for the pages to be generated. Next.js then calls **getStaticProps** for each id, which creates the props and passes it to the Component that renders the static HTML page for this id.

In order to see all the statically generated pages, first run “npm build” to build the project, and then “npx next export” to export the static pages to the “out” directory. Go to this directory and see all the generated HTML pages (and the associated vanilla JS for interactivity, e.g., carousel, multi-tab controls, CSR, etc.). We can now serve these pages using a standard Web server. In our case, simply run “serve out –p 8000” to start a Web server that runs at port 8000 to serve the directory “out”.

**Other Next.js Examples**

Here are other Next.js examples that you can look at:

15-NextJs-Astro/next-user-list-netninja

15-NextJs-Astro/nextjs12-posts-traversy

15-NextJs-Astro/nextjs12-blog

15-NextJs-Astro/nextjs13-ex

**Nuxt.js, Sveltekit**

While Next.js is an extension of React.js, other front-end frameworks have similar extensions:

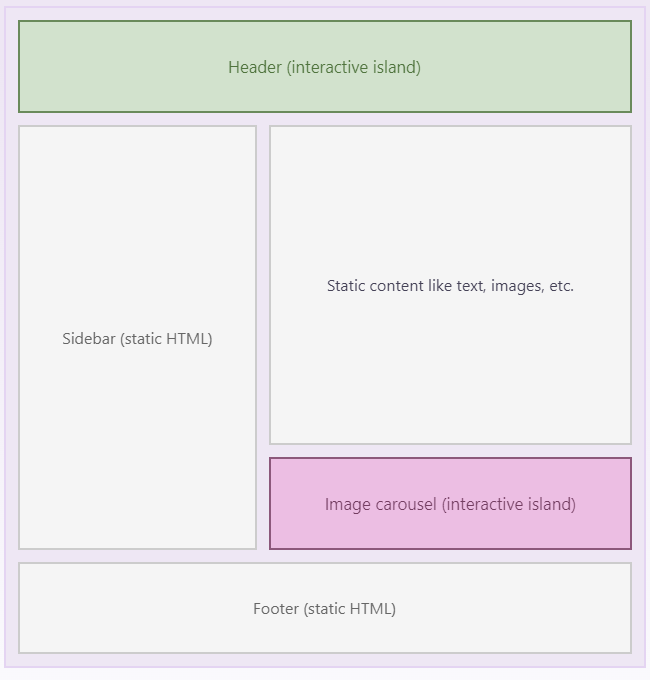
For Vue, it is Nuxt.js (<https://nuxtjs.org/>)

For Svelte, it is Sveltekit (<https://kit.svelte.dev/>)

**Astro.js (**<https://astro.build/>**)**

If your goal is mostly **Static Site Generation (SSG)**, there are other more suitable tools for this task: 1. **Astro** (<https://astro.build/>), 2. **Gatsby** (<https://www.gatsbyjs.com/>)

Astro is a JS framework and a great tool for **SSG**. It is an all-in-one web framework for building fast, content-focused websites. Its Web page has the following statement: “Pull content from anywhere and serve it fast with Astro's next-gen island architecture.” The idea is to pull data from a CMS/DB/Rest API (a list of recipes, a list of blogs, etc.), and dynamically generate static web pages (HTML+JS). It makes use of what is called the island architecture as shown below (<https://docs.astro.build/en/concepts/islands/>).



You can think of each island to be a Component. Components can be rendered using SSG to convert them into static HTML content during build time, or can be rendered at the client (CSR).

The best thing about Astro is that you can either write your own Astro components, which are very similar to React components, or use components from other front-end frameworks such as React, Vue, Svelte, etc. After implementing your Astro project, you can then use Astro build tools to convert your code into plain HTML/JS files, which are then stored in the static folder of a Web server and then served to the clients.

We should note that Astro generates a separate HTML/JS file for each route (page) and employs the Multi-Page Application (**MPA**) architecture as opposed to the **SPA**, which is employed by Next.js. Thus, when you go a different page in an Astro app, the page is completely re-loaded from scratch. This means that you can **NOT** share state across different pages of your app at the browser. Recall that since Next.js employs the SPA model, we were able to share the same state across different pages of our app using the React Context API. But it is possible to add a state manager to an Astro app and share state across different components of the same page. For details, refer to the Astro documentation. Here is Astro documentation on why and when you should use Astro: <https://docs.astro.build/en/concepts/why-astro/>

**Creating a simple Astro App**

To create a boilerplate Astro app, open npm and type:

|  |
| --- |
| % npm create astro@latest <the name of your project, e.g., AstroEx1 > |

Go to your project file, open it with VS code. Then type

|  |
| --- |
| % npm run dev |

This will compile your project and start serving it in a live server.

When you are done with your project, you can create the final static pages by running:

|  |
| --- |
| % npm run build |

This will create ALL static pages under the “dist” directory. You can then use a static page Web server to start serving these pages. For example, install “serve” utility, and start serving the Astro dist directory as follows:

|  |
| --- |
| % serve dist –p 8000 |

Here is an example with a single root page: 15-NextJs-Astro/AstroEx1

Here is another example that has multiple pages: 15-NextJs-Astro/AstroEx2

Here is an example that contains Astro, React, Vue, and Svelte components: 15-NextJs-Astro/AstroEx3. Notice that some of the React, Vue and Svelte components are rendered with ‘client:load’ so that they become reactive and rendered at the browser (CSR). Thus you can interact with these component at the browser and change them using JS.

For more Astro examples, please refer to: <https://github.com/withastro/astro>. Notice that it is also possible to enable server-side rending and deploy your app inside a Node.js adapter (<https://docs.astro.build/en/guides/server-side-rendering/>) such as Node.js, Netlify, Vercel, Cloudflare etc., but this is not what Astro is intended for. You should always keep in mind that Astro is a MPA and is intended for SSG using mostly non-changing static data. But it is possible to make use of CSR and SSR with Astro. For details, look at Astro documentation (<https://docs.astro.build/en/getting-started/>).