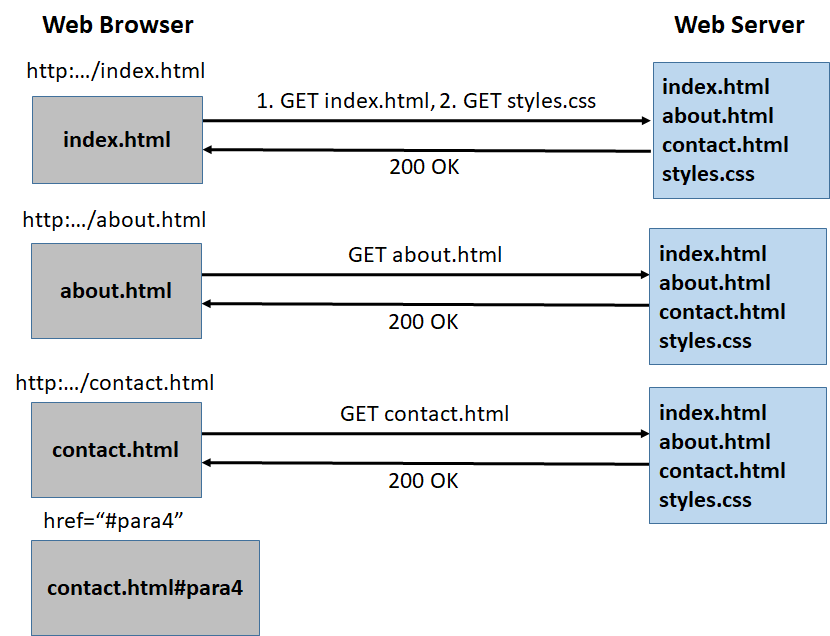
**History API & SPA Routing**

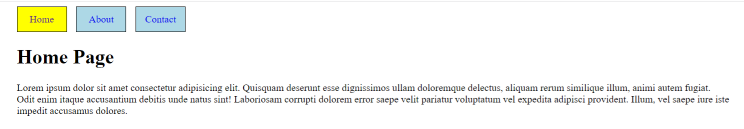
(<https://developer.mozilla.org/en-US/docs/Web/API/History_API>)

So far all our SPAs consisted of a single URL, the root URL ‘/’. We rendered all our SPA components under this root URL. But in most non-trivial applications, you will have many different URLs. For example, you may have a URL for the about page (/about), another for the contact page (/contact) etc. In a multi-page application, each of these pages will be stored in a separate HTML file and served from the Web server through different requests.

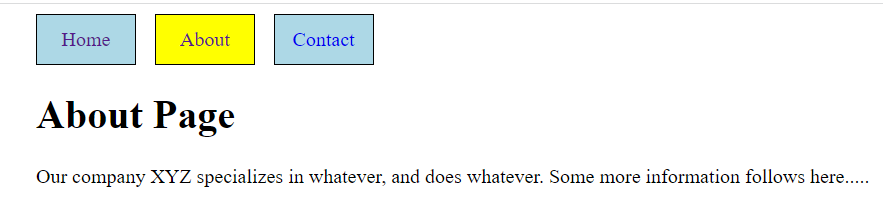
Here is an example that shows how things work in a Multi-Page Application (MPA): 11-Routing/01-MPA



We have 3 pages called index.html, about.html, contact.html. Initially, the user requests the root page (/ or /index.html). The browser sends an HTTP request to the server to get index.html. After downloading index.html, the browser starts parsing the HTML file and realizes that it needs to get style.css (because of the <link href=”…”> element). It then sends a second request to the Web server to get styles.css. After getting both the html file and the CSS file, the browser now renders the page on the screen. Here is how that page looks like:



As you can see, we have a navbar at the top that lets us go back and forth between the pages (home, about, contact). These are simply <a href=””> tags that lets the browser to go to a new page. The current (active) page link has a yellow background in the navbar. Assuming that we now want to go to the “About” page, we press the “About” button at the top. As you can see from the source of index.html, this is a <a href=”about.html”>About</a> link. To go to the new page, the browser sends a new request to the Web server and asks for “about.html”. The Web server sends about.html, and the browser now renders that page on the screen. Notice that the browser did not request styles.css this time because it was already cached in the local cache of the browser and has not changed since the last download. We can see this interaction between the browser and the server by opening the browser debug tools and going to the Network tab. Here is how the About page looks like. You can see that the About button in the navbar now has a yellow background indicating that it is the current page.



Finally, we press the Contact button at the top, which lets us go to the Contact page. Again, the browser sends a new request to the server, fetches contact.html, and renders its contents on the screen.

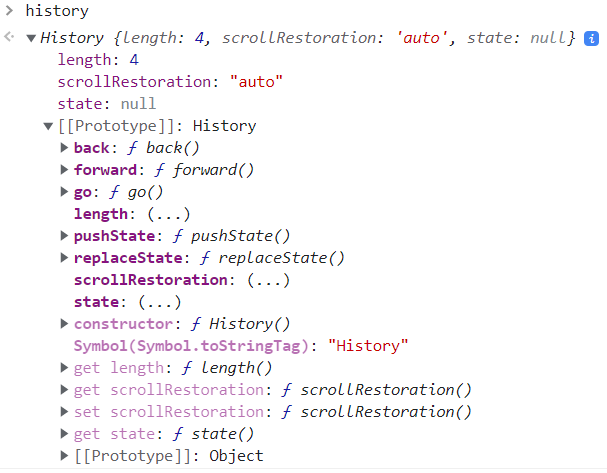
In contact.html, we have a link that does not take us to a new page, but rather to a new location in the page. To do this, we give an id to an html element (in this case, we give an id=”para4” to our 4th paragraph element), and specify the link as href=”#para4”. When we press this link, the browser automatically scrolls the page and takes us to that specify element so that we can see it. Also notice that since we are still inside the same page, no new HTML request is made to the server. Also notice the URL at the top: “http://.../contact.html#para4”. This is called a hash based URL.

07-Ajax/02-MPA-NetNinja-BlogPostApp-VanillaJS is a BlogPost app by Net Ninja implemented as a MPA in vanilla JS that uses the json-server to create, read, delete blog posts. Before running this app, make sure that you start json-server at the default port 3000 with data/db.json as the database. When you look at the code, you can see that “Add new blog” button on the main page is linked to a new page named “create.html” using the <a> tag. This means that when we press that link, the browser contacts the Web server and loads a new page called create.html. Similarly, “Read more” links below each Blog post is also linked to a new page called “details.html”. The id of the blog post is passed within the search parameters of the URL to the new page. The JS code (details.js) then takes this id from the URL, contacts the json-server to get the details of the blog and then renders it on the screen. But we can see that a new page is loaded from the server. Finally, when we are inside create.html or details.html, we go to the root page (/) by calling **window.location.replace**(“/”), which forces the browser to reload the root page from the Web server. Notice that although the data in the page is downloaded from the json-server by the respective JS code and then rendered in the page, this is a MPA because each time we go to a new route (page) in the app, the browser reloads the page from the Web server similar to the way we described above.

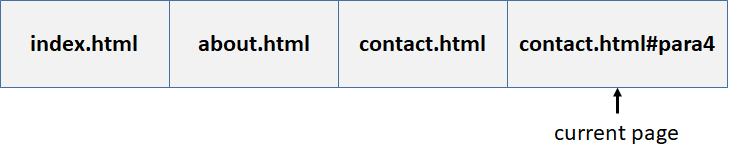
**History API (**<https://developer.mozilla.org/en-US/docs/Web/API/History>**)**

window.history (history for short because we are running inside a method that belongs to the window object) allows manipulation of the browser session history; that is, the pages visited in the tab or frame that the current page is loaded in. It keeps track of the history of the pages visited so far in a list (an array) and lets us move over this array using its .back(), .forward() or .go() methods. To see the details of the history object, open the developer tools, console, and type history. Here is how the output should look like after the sequence of operations we have performed above:

1. Load the root page (index.html)
2. Go to about.html
3. Go to contact.html
4. Go to paragraph 4 inside contact.html



You can see that the length of the history is 4 because we have visited the following 4 links (pages):



We can get more information about the current URL location we are in by looking at the window.location object. Go to the console and type > location and then expand the output to see its internal details. Here is how it looks in my browser:

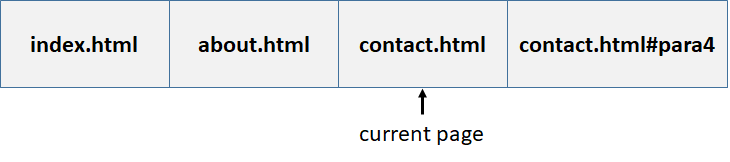


We can see that our current pathname is /contact.html, our hash value is “#para4”. There are a lot of other information about the host, full URL, origin, port, protocol, etc.

**Moving over the history**: We can go back and forth over the pages in our history by pressing the “Back” and “Forward” buttons of the browser seen at the top:



For example, when we press the “Back” button (<-), we go back one cell to the left on the history, which now looks like this:



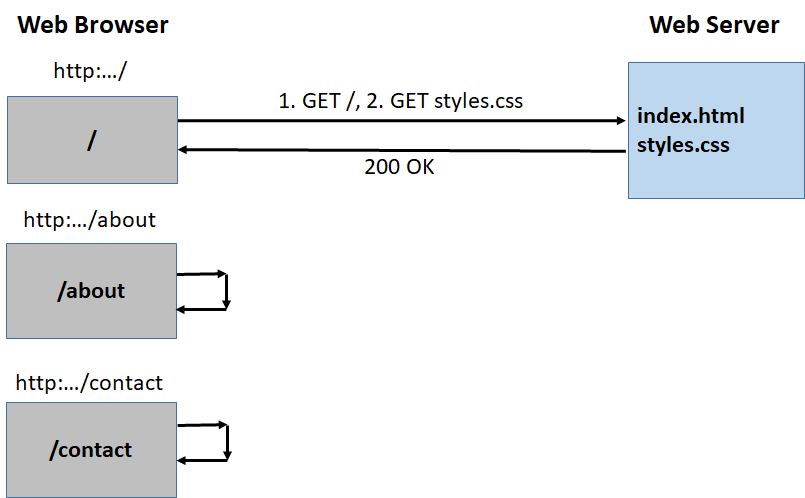
We can also go back/forward using JS (or from the console), but making calls to the back/forward/go methods of the history object. To try that out, open the console and type > history.back()

You will now see that you have moved one page to the left in the history and that the browser is showing “about.html” on the screen. You can also go multiple pages back and forth by using the history.go() method. For example, history.go(2) will move you 2 pages to the right (forward), and history.back(-2) will move you 2 pages to the left (back).

**onpopstate** event: It turns out that an event named “**onpopstate**” fires each time we press the back/forward buttons of the browser or call history.back()/history.forward()/history.go(). 11-Routing/03-HistoryAPI/index.html shows a simple app that makes use of the history object’s methods to dynamically render different routes (pages). To run this example, you need to open it in VS Code and serve it with live-server. You can see that when we call pushState(), a new page is added to the stack of pages in the history. We can also replace the state of the current page using replaceState(). This call does not push a new route into the history, rather, it changes the contents of the current route in the history. Notice that each time we call pushState and replaceState, the URL at the browser location tab changes. Also note that each time we press the back/forward buttons at the top, a ‘popstate’ event is generated. You must therefore install a handler for window.onpopstate and render the current page within that callback.

**Routing in a Single Page Application (SPA)**

We know that in a SPA, the app (or its root page, i.e. ‘/’) is loaded just once. After this, we never go to the Web server to load a new page. If we go to a new route (page) such as /about or /contact, the page must be rendered by JS within the browser without the browser ever sending a new request to the server. This is illustrated in the following figure:



You can see that the initial request to the root URL (/) goes to the Web server, which sends the root page (index.html). The browser then parses this file and downloads any CSS or JS files that have been included in this root file. It is usually the case that this page is empty, and contains a single root div element where JS frameworks such as React will render all routes inside the browser. After the initial root page is rendered, we usually have a navbar similar to the one we have shown in the MPA with links to other routes such as /about, /contact etc. When the user presses any of these links to go to those pages, however, we should somehow intercept that request and prevent it from going to the server. Instead, we should see where the request goes to and then render that page inside the browser. So, after the root page is downloaded from the server during the initial HTTP request, we should never go the Web Server to load any other HTML page. Obviously, we can contact the Web server REST API endpoints to get/post/put/patch/delete data using the fetch API and then update our page accordingly. But this is NOT a re-load of the page. This is the difference between an SPA and a MPA.

**How to perform Routing inside a SPA**

Before we talk about how we can perform routing inside a SPA, let’s first talk about the two types of URLs that the developers use in SPAs:

1. **Normal URL**: This is the URL that we are used to seeing, with normal path, like this:

[https://example.com/](https://example.com/https:/example.com/about)

[https:/example.com/about](https://example.com/https:/example.com/about)

<https://example.com/contact>

This kind of URL has the advantage of Search Engine Optimization (SEO), but you have to create many back-end endpoints (as many as application routes) in order to avoid the 404 error code message while loading the URL. These are the best-looking URLs, but they require your server to be configured correctly. Specifically, your web server needs to serve the same page at all URLs that are managed by your client-side SPA. When we cover React-Router, we will see that this is achieved by the <**BrowserRouter**> component.

1. **Hash Based** **URL:** This URL has a unique application endpoint controlled by one single back-end action and multiple hashtags managed by JS that define the SPA routes.

<https://example.com/#/about>

<https://example.com/#/contact>

In this case we just need to create one single controller for the app, which makes it possible to keep the whole structure clear. Although the SEO is more difficult to manage in this technique, it does not require any special set up at the Web server. When we cover React-Router, we will see that this is achieved by the <**HashRouter**> component. The problem with this technique is that the routes do not look very nice.

In this section we will see how we can implement a Router in vanilla JS that can serve multiple endpoints using the Normal URL technique. Our implementation will require that the user first enters our app from the “/” page and then navigate using our navigation links, but that should give you a general idea on how a router is implemented in vanilla JS.

11-Routing/04-JS-Router/index.html shows an SPA that serves 3 routes: /, /about and /contact. The idea is to specify the links to the routes using a custom-defined attribute named ‘**data-route**’ and then installing an onClick handler for each route. Inside the onClick handler, we first look at the current route and then nagivate to that page. That is, we dynamically render the page contents using JS. We never contact the Web server during route changes. This is what we mean by intercepting the request for the new route and then rendering it by JS.

11-Routing/05-BlogListApp shows another SPA using vanilla JS to process blog posts that uses history.replaceState to show the current route we are in: /, /create or /details. In this application, I preferred not to use history.pushState because I did not want the user to go back and forth using the Back/Forward buttons of the Browser. Rather, I wanted the user to interact with the application using the navigation buttons displayed by the browser. This prevents some errors that might otherwise occur and makes life easier while still displaying the current route we are in.

11-Routing/06-MyNoteCardAppWithMaterializeCSS shows yet another SPA using vanilla JS to process note cards. This time I used the Materialize CSS to make the interface look a little better. But the overall structure of the app is very similar to the BlogListApp. This time our routes are /, /create, /edit. Also notice that the onClick handlers for the routes are directly inserted into the HTML code. This code also uses history.replaceState to override the current route.

**React Router [**https://www.typescriptlang.org/**]**

While a simple React app (our Todo app) may consist of a single root ‘/’ route, where everything is handled, any non-trivial app will have many routes. For example, we may have a home page (/), an about route, and a contact route displaying different information. To handle such multiple routes, we need a Router. The current version of the React router is v6, but we will start with v5 (<https://v5.reactrouter.com/>) since most of the existing codebases still use this version and therefore, it is important that you get an understanding of v5 before going to v6.

**BrowserRouter vs HashRouter**

When we talked about implementing a router in JS, we said that it is possible to implement “normal” routes that look like /, /about, /contact or “hash” routes that look like /#/about, /#/contact. In React, there are two routers: BrowserRouter that implements the normal routes-based routing and HashRouter that implements hash-based routing. The first thing you need to do when implementing routes is to pick a router and enclose everything inside that router component. Below we show how we implement 3 routes using the BrowserRouter (Look at 11-Routing/07-Router5).

|  |
| --- |
| <BrowserRouter>        <div className="container">          {/\* Links to the routes \*/}          <div className="navbar">            <Link style={linkStyle} to="/">Home</Link>            <Link style={linkStyle} to="/about">About</Link>            <Link style={linkStyle} to="/contact">Contact</Link>          </div>          <Switch>            {/\* Three different ways to declare and render routes \*/ }            <Route              exact              path="/"              render={props=>(                <div>                  <h1>Here is a list of current items:</h1>                  <Home items={items}/>                </div>              )}            />              <Route path="/about">              <About name="XYZ"/>            </Route>            <Route              path="/contact"              component={Contact}            />            {/\* Redirect /oldroute to the root route / \*/}            <Route              path="/oldroute">                <Redirect to="/"/>            </Route>            {/\* Default route: 404 \*/}            <Route              path = "\*"              render={props=> (                <div>                  <p>404: Page not found</p>                  <br/>                  <Link to="/">Press here to go home</Link>                </div>                )}            />          </Switch>        </div>      </BrowserRouter> |

As you can see, the entire application is wrapped with <BrowserRouter> tags. All routes must be wrapped inside a <Switch> component and each individual route must be wrapped inside a <Route> component. Here we have five routes as specified by the “path” property. 3 of these, /, /about, /contact are valid routes. The fourth route, /oldroute, is an obsolete route, and simply redirect the user to the ‘/’ page using the <Redirect> component. Finally, the last route “\*” matches everything else and simply displays a 404 page not found error. Then displays a Link so that the user can go to the root page.

There are 3 ways to render what is to be rendered when we go that route as illustrated in the code:

1. Give the render function using the “render=” prop
2. Directly write the JSX inside <Route> </Route> tags
3. Give the name of the component using the “component=” prop

You can also try and see how HashRouter works by simply changing BrowserRouter to HashRouter. You will see that the internal working of the app will not change, but the routes will.

You should also notice that you can create a link that takes to a particular route in your app using the Link component as seen at the top of the code in the navbar component.

Another thing in that inside the <About> component, we created a button that takes us to the ‘/contact’ page when pressed. To do this programmatically, you use the useHistory hook as follows:

|  |
| --- |
| import React from 'react'  import { useHistory } from "react-router-dom"  export default function About({name}) {      const history = useHistory()      return (          <div>              <h1>This is our About page.</h1>              <p>Our company {name} specializes in whatever, and does whatever.                 Some more information follows here.....              </p>              <br/>              {/\* This is how you would go to a new route programmatically \*/}              <button onClick={()=>history.push('/contact')}>Go to contact page</button>          </div>      )  } //end-About |

Now, when the user presses “Go to contact page” button, history.push(‘/contact’) code will take us to that page.

**Upgrading to v6 (**[**https://reactrouter.com/en/v6.3.0/api**](https://reactrouter.com/en/v6.3.0/api)**)**

To upgrade your app to use React Router v6, you first need to install v6 as follows:

|  |
| --- |
| % npm install react-router-dom@6 |

The first thing we must do is to get rid of the Switch component and replace it with Routes component. Next, we need to change the way Route components are rendered. Instead of “component=” prop or “render=” prob, we now have “element=” prop in v6, where we write JSX code to render one or more components for the route. Finally, instead of using the Redirect component, we now use the <Navigate> component. The final code implemented using React Router v6 is given below (11-Routing/08-Router6):

|  |
| --- |
| <BrowserRouter>        <div className="container">          {/\* Links to the routes \*/}          <div className="navbar">            <Link style={linkStyle} to="/">Home</Link>            <Link style={linkStyle} to="/about">About</Link>            <Link style={linkStyle} to="/contact">Contact</Link>          </div>          <Routes>            {/\* Three different ways to declare and render routes \*/ }            <Route              exact              path="/"              element={(                <div>                  <h1>Here is a list of current items:</h1>                  <Home items={items}/>                </div>              )}            />              <Route path="/about"              element={<About name="XYZ"/>}            />            <Route              path="/contact"              element={<Contact />}            />            {/\* Redirect /oldroute to the root route / \*/}            <Route              path="/oldroute" element={<Navigate to="/"/>} />            {/\* Default route: 404 \*/}            <Route              path = "\*"              element = {(                <div>                  <p>404: Page not found</p>                  <br/>                  <Link to="/">Press here to go home</Link>                </div>              )}            />          </Routes>        </div>      </BrowserRouter> |

As for navigating to another page when a button is pressed, we will now use “useNavigate” instead of the “useHistory” hook. Here is the new <About> component:

|  |
| --- |
| import React from 'react'  import { useNavigate } from "react-router-dom"  export default function About({name}) {      const navigate = useNavigate()      return (          <div>              <h1>This is our About page.</h1>              <p>Our company {name} specializes in whatever, and does whatever.                 Some more information follows here.....              </p>              <br/>              {/\* This is how you would go to a new route programatically \*/}              <button onClick={()=>navigate('/contact')}>Go to contact page</button>          </div>      )  } //end-About |

But for a more complex app with sub-routes, there will be more changes. To see such an example, look at Net Ninja’s React Router 6 Tutorial playlist (<https://www.youtube.com/playlist?list=PL4cUxeGkcC9h7F1LWaQ7MAI8ptg5VjvxJ>), where he first gives you a React app implemented using v5 (11-Routing/09-NetNinja-Router5), and then converts this app to use React Router v6 (11-Routing/10-NetNinja-Router6).

**Net Ninja BlogListApp in React using Router v5 and JSON-Server**

We are now pretty much through learning React. For a complete tutorial teaching and using all of the concepts that we covered so far, follow Net Ninja’s Full Modern React Tutorial: <https://www.youtube.com/playlist?list=PL4cUxeGkcC9gZD-Tvwfod2gaISzfRiP9d>. There, he develops our famous BlogListApp in React using React Router v5 and json-server.

Here is the code: 11-Routing/11-NetNinja-BlogListApp

To run this application, first run the json-server at port 8000 by typing “% json-server data/db.json –port 8000”. Then start the app using “% npm start”.