Declarative routing with web components

Jason Strimpel
@jasonstrimpel

The history API in the browser changed the way how the web application model was developed. Prior to the history API developers had to resort to using server-side. With the arrival of history API we now are able to spin up single page content and its functionality according to the URL updates. This changed the way developers thought about how to be crafted and now the browser can take care of entire routing over the page load.

Nowadays most of the front end frameworks provide their own routing implementation. Some of them are very powerful and provide a lot of commands. Since the history API in browsers work in a way that fires the component to the URL and does the update on the application view, state according to the URL.

In this article, we are going to look at how we can create a client-side router from scratch using vanilla javascript and custom elements. In our case using custom elements we should be able to create declarative routing elements which can be used in the markup. In order to do so we need to extend the base custom elements, the syntax should be familiar to you. The exact router provides custom components with the name *route* and *route-link*. *route* is called over the *route* tag. Here let's use the same concept but with custom element extend syntax. Let's begin with the markup.

The Markup

Create a file called index.html with the below content.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="author" content="Jason">
    <meta
    name="description" content="">
    <meta
    name="viewport" content="width=device-width, initial-scale=1">
    <meta
    name="theme-color" content="#ffffff">
    <title>Vanilla Router</title>
    <style>
      .route {
        display:
        none;
        opacity:
        0;
        transition:
        opacity
        ease-in-out
        .3s;
      }
      .route-active {
        display:
        block;
        opacity:
        1;
      }
    </style>
    <script type="module" src="./router.js">
    </script>
  </head>
  <body>
    <header>
      <nav>
        <route-link
        path="/">Home
        </route-link>
        <route-link
        path="/about">About
        </route-link>
        <route-link
        path="/contact">Contact
        </route-link>
      </nav>
    </header>
    <main>
      <route
      path="/">This is the
      home page.
      </route>
      <route
      path="/about">This is the
      about page.
      </route>
      <route
      path="/contact">This is the
      contact page.
      </route>
    </main>
    <script>
      customRoutes.define("no-route", Route);
    </script>
  </body>
</html>
```

The above code does nothing with the implementation script section. Let's check the element syntax before going to the script section. The **<nav>** section has two **<route-link>** elements with the attribute **path**. Which implies for routing. All *route-link* elements are wrapped in **<header>** tag. The tag name should comply with the web component standard and you can use any name as prefix instead of *route-link*. In our **<header>** section **<main>** is having three attributes path, title, and component where the path and component attributes are required. The path attribute specifies the URL path that should be matched against the browser history updates. The component attribute specifies the name of the custom element that should be updated when changing the route.

You could see one more tag called **<no-route />** inside the **<main />** tag. Here, this is the element where the history object state is changed but no matching route is found. In this case the component specified in the *no-route* will be rendered in the **<no-route />** element. If the site path contains any dynamic parameters, it will be extracted from the URL and will be passed as the attribute to the matched component.

The browser will never throw an error if the page contains invalid elements. So the above markup should be rendered in the browser without any errors and nothing code for the custom element which will take care of the routing in our scenario. We need to write the code for the custom elements to show markup as those are used as placeholders and data structure.

Router element

Create a file called router.js and add the following code in it.

```
import defaultClass from./utils/utils.js;
import {Route}from./route.js;
import {log}from./utils/log.js;

customElements.define("no-route", Route);
```

The above code does nothing other than registering the router element to the custom element registry with tag name **no-route**. Now we are using utils and tag name of our router from the **utils** and **route** files in your HTML file.

```
<script type="module" src="./router.js">
```

The attribute **type="module"** is important as we are using ES Modules. Use any simple HTTP server to run the file. You can use Node's **http-server** module to run your site. Now if you open the browser and access the HTML file via the localhost, you should be getting the message in your developer tools console.

Building the routes definitions

Each individual route is defined by a **<route />** tag. It's the route tag's attributes that specify how the URL path should be matched against the browser's URL. We should also watch for route changes to update the corresponding component. Add the following code in the Route class.

```
import defaultClass from./utils/utils.js;
import {log}from./utils/log.js;

get isActive() {
  return this.getAttribute("isActive");
}

Add the above code in the Route class. As mentioned above, the DOM
change is triggered when the attribute **isActive** is set. We have added a
property which gives the reference to the outer element as we have to use
the outer element very frequently.

Navigating with router

Once the page is loaded and the Router is added to the DOM tree, we need
to fire the initial route change event. We need to listen to the browser's history
and notify about the route changes. So the router needs to check the
registered routes to get the suitable route to display it or it should display
the default route if no matching route found.

So let's add a new method called **navigate**. The navigate method should be
called once the Router element is loaded so that the default route level
components can be updated to the current URL.

history.pushState(data, title, path);

The **pushState** method accepts a simple Javascript object as the first
parameter that can be associated with the new history entry. The
parameter is the pattern that we are going to navigate. For detailed
documentation please refer to Mozilla Docs
https://developer.mozilla.org/en-US/docs/Web/API/History/pushState

Matching the component

When the history object state is changed, the router needs to look for any
matching routes to update the view. Create a function called **match** with
the following content.

let parseRx = /:(\w+)/g;

function compileRoute(path) {
  return new\ RegExp("^"+path.replace(/\*/\*\*\*\*/g, "*\*\*\*\*").
}

function matchRoutes(routes, path) {
  let\ params\ =\ {};\ \ \ \
```