SSR

Server Side Rendering

Problem

- SPA initial load can be slow
- SEO

53% of mobile site visits are abandoned if pages take longer than 3 seconds to load. (angular docs)

How can we improve those important problems that we have with Single Page Applications?

Without Server Side Rendering (SSR)?

Usually in SPA the initial HTML we get from the server looks similar to this:

```
<html>
...
<body>
        <script src=”my-spa-app.js”></script>
</body>
</html>
```

- the body is empty and contains a download of a script
- the script is running and in charge of presenting the page to the user
- initial load can be slow - we need to grab the html and js and only after that we need to run the js to render the page
- Unfriendly to search engines

With SSR

- with server side rendering the initial HTML we get from the server looks like this:

```
<html>
    ...
    <body>
        <h1>The full page is rendered by the server</h1>
        <p>we get a full html page like it was when there was no SPA</p>
        <p>We can also use server caching so the html will be sent faster</p>
        ....
        <script src=”my-spa-app.js”></script>
    </body>
</html>
```

- the first html is sent by the server, after that the spa takes control and everything is loaded dynamically
- modern frameworks/libraries like angular2/react support SSR - we have to make sure that the code we write is universal

Our first SSR

- Let’s create a new @angular/cli project
- run:
  - **ng serve**
  - this will start a development server **webpack-dev-server**
  - the server will auto refresh when you change the src code
- right click the page and choose: **View Page Source**
- notice the the development server does not server side render the app
- the development server can’t be used in production
- Let’s try and create a simple server to serve our app in production

How SSR works

- Server takes index.html file
- Server turns the angular components to HTML and place them in the **app** tag in the html
- Server will turn the components to HTML based on the route
- Server will not support browser events and browser specific api’s
- You can consider the server page as a quick landing page and after the angular script are downloaded and run we will hide the server page and present the page the browser created
- The server created page will pass support for the dynamic page created by the browser

@angular/cli universal support

- If your app is created with @angular/cli, adding universal support is a breeze
- @angular/cli has a new feature that will create SSR configuration for you
  - **> ng g universal universal-app-name**
- the command will do the following:
  - create **app.server.module.ts** with the **AppModule** wrapped in a **AppServerModule**
  - create entry point file **main.server.ts**
  - add the proper configuration in the **.angular-cli.json** for the server run by the backend
  - create a **tsconfig.server.json** file with the typescript configuration for the server app
  - install additional npm packages needed to run the app in the server
  - modify the **main.ts** to load the content for the browser after the dom is finished
  - modify the **app.module.ts** to add **BrowserModule.withServerTransition**
- run the command: **> ng build && ng build --app=universal-app-name**

ServerModule

- in the file: **app.server.module.ts** a server module is created
- the server module wraps your app and contains low level tools that can work in the server side as well
- the module will import the following
  - AppModule
  - ServerModule
- that module will be the root module for the server
- you can also add providers that will only be used in the server side

renderModule

- **renderModule** can turn our **AppServerModule** to **HTML** string
- as first argument we give the module (**AppServerModule**)
- the second argument is options dictionary containing the url and document
- let’s use the **renderModule** to create a simple js file that will be run by our server

server.js

```
require('zone.js/dist/zone-node');

require('reflect-metadata');

const { AppServerModule } = require('./dist-server/main.bundle');

const { renderModule } = require('@angular/platform-server');



renderModule(AppServerModule, {

   url: '/',

   document: '<app-root></app-root>'

}).then(html => {

   console.log(html);

});
```

ngExpressEngine

- express engine can turn static files like html templates, to generated html string
- **ngExpressEngine** is an express engine that can make html from angular app
- **ngExpressEngine** accepts an option dictionary with the module we want to bootstrap (in our example **AppServerModule**)
- using express engine we want the end result that whenever we use **res.render** on html files express will use **ngExpressEngine**
- let’s change the server file and create an express app and load in our express app the **ngExpressEngine**
-

ngExpressEngine

```
//create the ngExpressEngine
app.engine('html', ngExpressEngine({
    bootstrap: AppServerModule
}));

// set the engine
app.set('view engine', 'html')

// set where are views are located, in our case where the index.html is
app.set('views', path.resolve(__dirname, 'dist'));
```

Setting the express route

- we want for every route to serve the index.html rendered by the engine we set
- we can create an express route with ‘*’ to catch everything
- we can use **Response.render** to render the html with **ngExpressEngine**
- **render** get’s the view as first argument and the second argument is options for the engine and in our case we need to pass the request

```
app.get('*', (req: Request, res: Response, next) => {
    res.render('index.html', {req});
});
```

- try to run your app again and click the view page source to examine the html sent from the server

static files

- our server will also have to serve the static files
- we can use **express.static(assetsDir)** to add a middleware for express that will serve static files
- how can we distinguish routes for files and routes for our app?

Good code Bad Code

- Our app is now universal and this requires a certain shift in the way we write code
- say we want to create a directive that will be placed on buttons and will add a loading spinner on buttons
- there is a jquery plugin that does that which is called **ladda**
- that plugin will require us to use **jquery**
- let’s install **jquery** and **ladda** and create our directive
- in the **.angular-cli** in the browser app you can add the scripts you installed
- you can use those scripts to create our directive
-

Ladda Directive

- in our constructor in the directive we will initiate ladda on the **ElementRef**
- the directive will accept an input with a setter that will activate the ladda based on the boolean input
- after creating our directive try build the app and run the universal app

Bad Code

```
@Directive({
 selector: '[nzLadda]'
})
export class LaddaDirective {
 @Input('nzLadda') public set isLoading(val: boolean) {
   if (val) {
     this._l.ladda('start');
   } else {
     this._l.ladda('stop');
   }
 }
 private _l: any;
 constructor(private _element: ElementRef) {
   this._l = $(this._element.nativeElement).ladda();
 }
}
```

Bad Code

ReferenceError: $ is not defined

Good Code

```
@Input('nzLadda') public set isLoading(val: boolean) {
 if (!isPlatformBrowser(this._platform)) {
   return;
 }
 if (val) {
   this._l.ladda('start');
 } else {
   this._l.ladda('stop');
 }
}
private _l: any;
constructor(private _element: ElementRef, @Inject(PLATFORM_ID) private _platform: Object) {
 if (isPlatformBrowser(this._platform)) {
```

Good code

- our code needs to run in the server side
- if we use api’s specific to the browser we need to branch our code for server and client
- to do this we can inject **PLATFORM_ID**
- we can use **isPlatformBrowser** or **isPlatformServer** to know where we are running now

Routing

- proper web app should render the correct page when we reload it
- routes should still work
- our server side should render the proper page when reloading the app and create a full html for that page
- let’s create another page called about page and check if our server can deal with routes

Some server side optimizations

- one of the reasons for SSR is increasing the initial load of our application
- now that the server renders the first view we can add additional optimizations for our app.

gzip

- helps decrease the size of the response body
- reducing the size of the body will increase the transfer speed of the resources to the browser
- gzip can be configured as an express middleware or at the reverse proxy level like in nginx
- let’s add express gzip middleware to our express app
  - **npm install compression --save**
  - **const compression = require(‘compression’)**
  - **app.use(compression())**

caching

- it’s better to handle caching with the reverse proxy
- this way a lot of the requests will not even pass to our express app
- this will save our express app to handle more important issues

Summary

- Server Side Rendering is a must for applications running in production
- our frontend app should run with a proper server not with the default dev server
- we need to make sure the code we write can be run on the server side as well