

Join GitHub today

[Dismiss](#)

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)

BuildOASP4JsApplication

Santos Jiménez Linares edited this page 22 days ago · 53 revisions

Table of Contents

- [Build your own OASP4Js application](#)
- [Goal of Jump The Queue](#)
- [Installing global tools](#)
 - [Visual Code:](#)
 - [Node.js](#)
 - [TypeScript](#)
 - [Yarn](#)
 - [Angular/CLI](#)
- [Creating basic new project](#)
- [Adding Google Material and Covalent Teradata](#)
- [Start the development](#)
 - [Creating components](#)
 - [Creating services](#)
- [Making calls to server](#)
- [Next chapter: Deploy your OASP4Js app](#)

[▶ Pages](#) 32

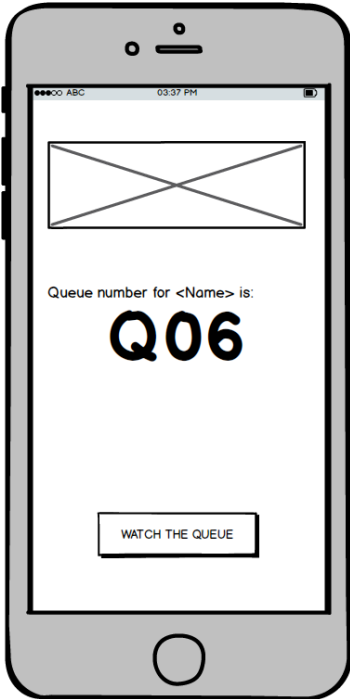
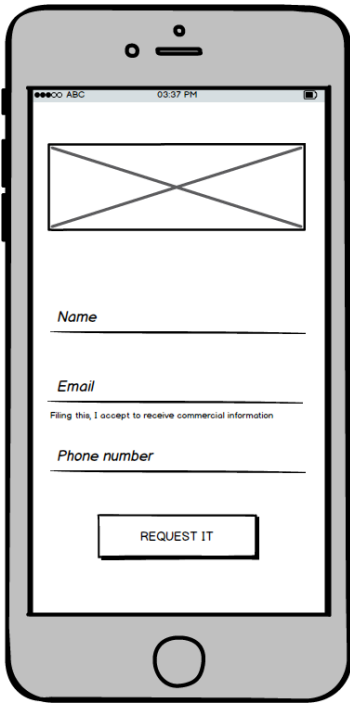
Build your own OASP4Js application

In this chapter we are going to see how to build a new OASP4Js from scratch. The proposal of this tutorial is to end having enough knowledge of Angular and the rest of technologies regarding OASP4Js to know how to start developing on it and if you want more advanced and specific functionalities see them on the cookbook.

Goal of Jump The Queue

This mock-up images shows what you are going to have as a result when the tutorial is finished. An app to manage codes assigned to queuers in order to easy the management of the queue, with a code, you can jump positions in queue and know everywhere which is your position.


- [Home](#)
- [OASP intro](#)
- [Jump The Queue](#)
 - [Jump The Queue Design](#)
- [OASP Backend Technologies](#)
 - [OASP4J Getting Started](#)
 - [OASP intro](#)
 - [Oasp4j overview](#)
 - [devonfw intro](#)
 - [An OASP4J application](#)
 - [Build your own OASP4J application](#)
 - [OASP4J application components](#)
 - [OASP4J layers](#)
 - [OASP4J adding custom functionality](#)
 - [OASP4J validations](#)
 - [OASP4J testing](#)
 - [Deployment](#)
 - [OASP4Fn Getting Started](#)
 - [OASP4Fn intro](#)
 - [An OASP4Fn application](#)
 - [Build your own OASP4Fn application](#)
 - [Testing](#)
 - [Deployment](#)
- [OASP Frontend Technologies](#)
 - [OASP4Js Getting Started](#)
 - [OASP4Js intro](#)

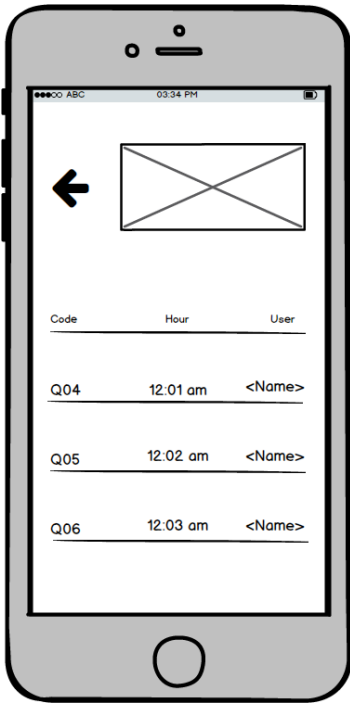


- [An OASP4Js application](#)
- [Build your own OASP4Js application](#)
- [The Angular Components](#)
- [The Angular Services](#)
- [Deployment](#)

Clone this wiki locally

<https://github.com/oasp/oas> 

 Clone in Desktop



So, hands on it, let's configure the environment and build this app!

Installing global tools

Visual Code:

To install the editor download the installer from [the official page](#) and install it.

Once installed, the first thing you should do is install the extensions that will help you during the development, to do that follow this steps:

1. Install Settings Sync extension.
2. Open the command palette (Ctrl+Shift+P) and introduce the command: **Sync: Download Settings**.

Provide GIST ID: [3b1d9d60e842f499fc39334a1dd28564](#).

In the case that you are unable to set up the extensions using the method mentioned, you can also use the scripts provided in [this repository](#).

Node.js

Go to the [node.js official page](#) and download the version you like the most, the LTS or the Current, as you wish.

The recommendation is to install the latest version of your election, but keep in mind that to use Angular CLI your version must be at least 8.x and npm 5.x, so if you have a node.js already installed in your computer this is a good moment to check your version and upgrade it if it's necessary.

TypeScript

Let's install what is going to be the main language during development: TypeScript. This ES6 superset is tightly coupled to the Angular framework and will help us to get a final clean and distributable JavaScript code. This is installed globally with npm, the package manager used to install and create javascript modules in Node.js, that is installed along with Node, so for install typescript you don't have to install npm explicitly, only run this command:

```
npm install -g typescript
```

Yarn

As npm, [Yarn](#) is a package manager, the differences are that Yarn is quite more faster and usable, so we decided to use it to manage the dependencies of Oasp4Js projects.

To install it you only have to go to [the official installation page](#) and follow the instructions.

Even though, if you feel more comfortable with npm, you can remain using npm, there is no problem regarding this point.

Angular/CLI

CLI specially built for make Angular projects easier to develop, maintain and deploy, so we are going to make use of it.

To install it you have to run this command in your console prompt: `npm install -g @angular/cli`

Then, you should be able to run `ng version` and this will appear in the console:

```
λ ng version

Angular CLI
-----
Angular CLI: 6.0.7
Node: 8.11.1
OS: win32 x64
Angular:
...

Package          Version
-----
@angular-devkit/architect    0.6.7
@angular-devkit/core        0.6.7
@angular-devkit/schematics  0.6.7
@schematics/angular         0.6.7
@schematics/update          0.6.7
rxjs                      6.2.0
typescript              2.7.2
```

In addition, you can set Yarn as the default package manager to use with Angular/CLI running this command:

```
ng config -g cli.packageManager yarn
```

Finally, once all these tools have been installed successfully, you are ready to create a new project.

Creating basic new project

One of the best reasons to install Angular/CLI is because it has a feature that creates a whole new basic project where you want just running:

```
ng new <project name>
```

Where `<project name>` is the name of the project you want to create. In this case, we are going to call it `JumpTheQueue`. This command will create the basic files and install the dependencies stored in `package.json`

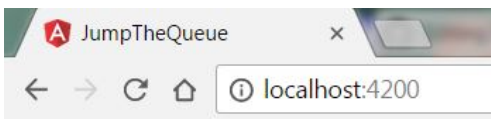
```
Successfully initialized git.
Installing packages for tooling via yarn.
Installed packages for tooling via yarn.
Project 'JumpTheQueue' successfully created.
```

Then, if we move to the folder of the project we have just created and open visual code we will have something like this:



Finally, it is time to check if the created project works properly. To do this, move to the projects root folder and run: `ng serve -o`

And... it worked:



app works!

Adding Google Material and Covalent Teradata

First, we are going to add **Google Material** to project dependencies running the following commands:

```
yarn add @angular/material @angular/cdk
```

Then we are going to add animations:

```
yarn add @angular/animations
```

Finally, some material components need gestures support, so we need to add this dependency:

```
yarn add hammerjs
```

That is all regarding Angular/Material. We are now going to install **Covalent Teradata** dependency:

```
yarn add @covalent/core@2.0.0-beta.1
```

Now that we have all dependencies we can check in the project's `package.json` file if everything has been correctly added (the following dependencies section is shown as it was at the time of writing this document):

```
"dependencies": {
  "@angular/animations": "6.0.3",
  "@angular/cdk": "6.2.1",
  "@angular/common": "6.0.3",
  "@angular/compiler": "6.0.3",
  "@angular/core": "6.0.3",
  "@angular/forms": "6.0.3",
  "@angular/http": "6.0.3",
  "@angular/material": "6.2.1",
  "@angular/platform-browser": "6.0.3",
  "@angular/platform-browser-dynamic": "6.0.3",
  "@angular/platform-server": "6.0.3",
  "@angular/router": "6.0.3",
  "@angular/service-worker": "6.0.3",
  "@covalent/core": "^2.0.0-beta.1",
  "core-js": "^2.4.1",
  "hammerjs": "^2.0.8",
```

```

"moment": "^2.20.1",
"rxjs": "^6.1.0",
"rxjs-compat": "^6.1.0",
"zone.js": "^0.8.26"
},

```

Now let's continue to make some config modifications to have all the styles and modules imported to use Material and Teradata:

- Angular Material and Covalent need the following modules to work: `CdkTableModule`, `BrowserAnimationsModule` and **every Covalent and Material Module** used in the application. So make sure you import them in the `imports` array inside of `app.module.ts`. These modules come from `@angular/material`, `@angular/cdk/table`, `@angular/platform-browser/animations` and `@covalent/core`.
- Create `theme.scss`, a file to config themes on the app, we will use one *primary* color, one secondary, called *accent* and another one for *warning*. Also Teradata accepts a foreground and background color. Go to `/src` into the project and create a file called **theme.scss** whose content will be like this:

```

@import '~@angular/material/theming';
@import '~@covalent/core/theming/all-theme';

@include mat-core();

$primary: mat-palette($mat-blue, 700);
$accent:  mat-palette($mat-orange, 800);

$warn:    mat-palette($mat-red, 600);

$theme: mat-light-theme($primary, $accent, $warn);

$foreground: map-get($theme, foreground);
$background: map-get($theme, background);

@include angular-material-theme($theme);
@include covalent-theme($theme);

```

- Now we have to add these styles in angular/CLI config. Go to `.angular-cli.json` to "styles" array and add theme and Covalent platform.css to make it look like this:

```

"styles": [
  "src/styles.css",
  "src/theme.scss",
  "node_modules/@covalent/core/common/platform.css"
],

```

With all of this finally done, we are ready to start the development.

Start the development

Now we have a fully functional blank project, all we have to do now is just create the components and services which will compose the application.

First, we are going to develop the views of the app, through its components, and then we will create the services with the logic, security and back-end connection.

Creating components

Note	Learn more about creating new components in OASP4Js HERE
------	--

The app consists of 3 main views:

- Access
- Code viewer

- List of the queue

To navigate between them we are going to implement routes to the components in order to use Angular Router.

To see our progress, move to the root folder of the project and run `ng serve` this will serve our client app in `localhost:4200` and keeps watching for changing, so whenever we modify the code, the app will automatically reload.

Root component

`app.component` will be our Root component, so we do not have to create any component yet, we are going to use it to add to the app the elements that will be common no matter in what view we are.


Note	Learn more about the root component in OASP4Js HERE
------	---

This is the case of a header element, which will be on top of the window and on top of all the components, let's build it:

The first thing to know is about [Covalent Layouts](#) because we are going to use it a lot, one for every view component.

Note	Learn more about layouts in OASP4Js HERE
------	--

As we do not really need nothing more than a header we are going to use the simplest layout: **nav view**

Remember that we need to import in `app.module` the main `app.component` and every component of **Angular Material** and **Covalent Teradata** we use (i.e. for layouts it is `CovalentLayoutModule`). Our `app.module.ts` should have the following content: 

```
// Covalent imports
import {
  CovalentLayoutModule,
  CovalentCommonModule,
} from '@covalent/core';

// Material imports
import {
  MatCardModule,
  MatInputModule,
  MatButtonModule,
  MatButtonModuleToggleModule,
  MatIconModule,
  MatSnackBarModule,
  MatProgressBarModule,
} from '@angular/material';
import { CdkTableModule } from '@angular/cdk/table';

// Angular core imports
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';
import 'hammerjs';

// Application components and services
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    BrowserAnimationsModule,
    HttpClientModule,
```

```

MatCardModule,          // Angular Material modules we are going to use
MatInputModule,
MatButtonModule,
MatButtonToggleModule,
MatIconModule,
MatProgressBarModule,
MatSnackBarModule,
CovalentLayoutModule,  // Covalent Teradata Layout Module
CovalentCommonModule,
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }

```

Note	Remember this step because you will have to repeat it for every other component from Teradata you use in your app.
------	--

Now we can use layouts, so lets use it on *app.component.html* to make it look like this: 

```

<td-layout-nav>          // Layout tag
  <div td-toolbar-content>
    Jump The Queue       // Header container
  </div>
  <h1>
    {{title}}           // Main content
  </h1>
</td-layout-nav>

```

Note	Learn more about toolbars in OASP4Js HERE
------	---

Once this done, our app should have a header and the "app works!" should remain in the body of the page:



app works!




To make a step further, we have to modify the body of the Root component because it should be the **output of the router**, so now it is time to prepare the routing system.

First we need to create a component to show as default, that will be our access view, later on we will modify it on it's section of this tutorial, but for now we just need to have it: stop the `ng serve` and run `ng generate component access`. It will add a folder to our project with all the files needed for a component. Now we can move on to the router task again. Run `ng serve` again to continue the development.

Let's create the module when the Router check for routes to navigate between components.

1. Create a file called *app-routing.module.ts* and add the following code: 


```

imports... 

const appRoutes: Routes = [ // Routes string, where Router will check the navigation and it
  { path: 'access', component: AccessComponent}, // Redirect if url path is /:
  { path: '**', redirectTo: '/access', pathMatch: 'full' }]; // Redirect if url path do not

@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes, {
        enableTracing: true
      }, // <-- debugging purposes only
    ),
  ],
  exports: [
    RouterModule,
  ],
})
export class AppRoutingModule {} // Export of the routing module.

```

Time to add this *AppRoutingModule* routing module to the app module: 

```

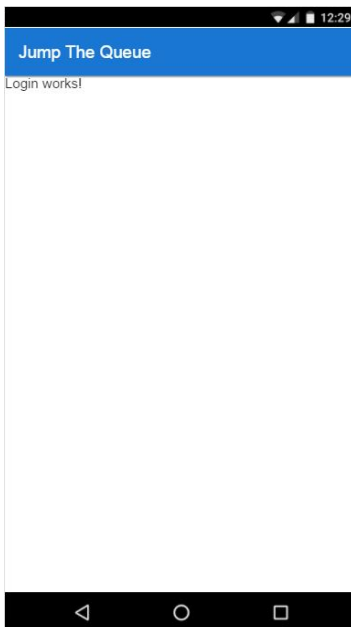
...
imports: [
  BrowserModule,
  AppRoutingModule,
  CovalentLayoutModule,
...

```

Note

Learn more about routing in OASP4Js [HERE](#)

Finally, we remove the "`{{title}}`" from *app.component.html* and in its place we put a `<router-outlet></router-outlet>` tag. So the final result of our Root component will look like this:



As you can see, now the body content is the html of **AccessComponent**, this is because we told the Router to redirect to Access when the path is `/access`, but also, redirect to it as default if any of the other routes match with the path introduced.

We will definitely going to modify the header in the future to add some options like log-out but, for the moment, this is all regarding Root Component.

AccessComponent

As we have already created this component from the section before, let's move on to building the template of the access view.

First, we need to add the Covalent Layout and the card:

```
<td-layout>
  <mat-card>
    <mat-card-title>Access</mat-card-title>
  </mat-card>
</td-layout>
```

This will add a grey background to the view and a card on top of it with the title: "Access", now that we have the basic structure of the view, let's add the form with the information to access to our queue number:

- Name of the person
- Email
- Telephone number

One simple text field, one text field with email validation (and the legal information regarding emails) and a number field. Moreover, we are going to add this image:



In order to have it available in the project to show, save it in the following path of the project: `/src/assets/images/` and it has been named: `jumptheq.png`

So the final code with the form added will look like this:

```
<td-layout>
  <mat-card>
    
    <mat-card-title>Access</mat-card-title>
    <form layout="column" class="pad" #accessForm="ngForm">

      <mat-form-field>
        <input matInput placeholder="Name" ngModel name="name" required>
      </mat-form-field>

      <mat-form-field>
        <input matInput placeholder="Email" ngModel email name="email" required>
      </mat-form-field>
      <span class="text-sm">Filling this, I accept to receive commercial information.</span>

      <mat-form-field>
        <input matInput placeholder="Phone" type="number" ngModel name="phone" required>
      </mat-form-field>

      <mat-card-actions>
        <button mat-raised-button color="primary" [disabled]="!accessForm.form.valid" class="t
      </mat-card-actions>

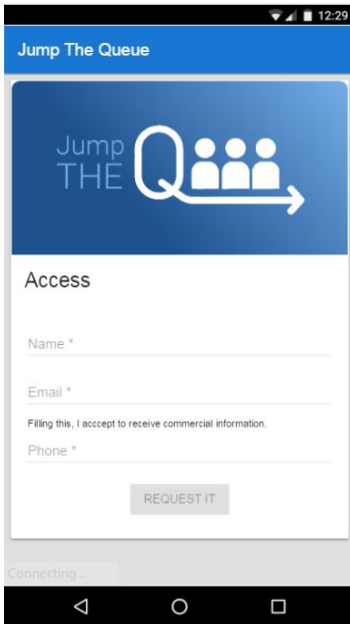
    </form>
  </mat-card>
</td-layout>
```

This form contains three input container from Material and inside of them, the input with the properties listed above and making all required.

Also, we need to add the button to send the information and redirect to code viewer or show an error if something went wrong in the process, but for the moment, as we neither have another component nor the auth service yet, we will implement the button visually and the validator to disable it if the form is not correct, but not the click event, we will come back later to make this working.

Note Learn more about forms in OASP4Js [HERE](#)

This code will give us as a result something similar to this:



Now lets continue with the second component: Code viewer.

Code viewer component

Our first step will be create the component in the exact same way we did with the access component: `ng generate component code-viewer` and we add the route in the `app-routing.module.ts`:



```
const appRoutes: Routes = [
  { path: 'access', component: AccessComponent},
  { path: 'code', component: CodeViewerComponent}, //code-viewer route added
  { path: '**', redirectTo: '/access', pathMatch: 'full' }];
```

With two components already created we need to use the router to navigate between them. Following the application flow of events, we are going to add a `navigate` function to the submit button of our access form button, so when we press it, we will be redirected to our code viewer.

Turning back to `access.component.html` we have to add this code:



```
<form layout="column" class="pad" (ngSubmit)="submitAccess()" #accessForm="ngForm"> // addec
...
<button mat-raised-button type="submit" color="primary" ... </button> // added type="submit"
```

This means that when the user press enter or click the button, `ngSubmit` will send an event to the function `submitAccess()` that should be in the `access.component.ts`, which is going to be created now:

```
constructor(private router: Router) { }

submitAccess(): void {
  this.router.navigate(['code']);
}
```

We need to inject an instance of `Router` object and declare it into the name `router` in order to use it into the code, as we did on `submitAccess()`, using the `navigate` function and redirecting to the next view, in our case, the code-viewer using the route we defined in `app.routes.ts`.

Now we have a minimum of navigation flow into our application, this specific path will be secured later on to check the access data and to forbid any navigation trough the URL of the browser.

Let's move on to *code-viewer* make the template of the component. We need a big code number in the middle and a button to move to the queue:

```
<td-layout>
  <mat-card>
    
    <mat-card-title>Queue code for {{name}} is:</mat-card-title> // interpolation of the var

    <h1 style="font-size: 100px" class="text-center text-xxl push-lg">{{code}}</h1> // queue

    <div class="text-center pad-bottom-lg">
      <button mat-raised-button (click)="navigateQueue()" color="primary" class="text-upper">
    </div>

  </mat-card>
</td-layout>
```

And the implementation of the *code-viewer.component.ts* should be something like:

```
imports...

export class CodeViewerComponent implements OnInit {

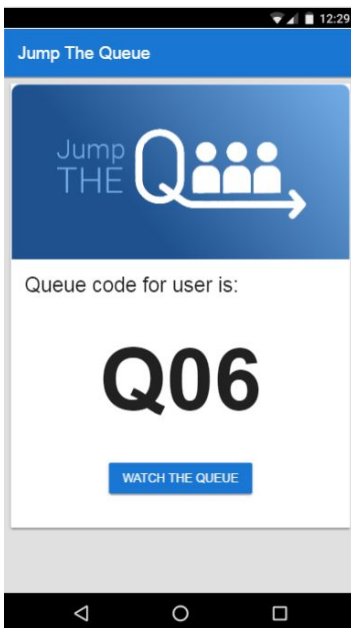
  code: string; // declaration of vars used in the template
  name: string;

  constructor(private router: Router) { } // instance of Router

  ngOnInit(): void {
    this.code = 'Q06'; //This values in the future will be loaded from a service maki
    this.name = 'Someone';
  }

  navigateQueue(): void {
    // this will be filled with the router navigate function when we
  }
}
```

Giving this as a result:



Finally, we are going to add an icon button to the header to log out, we are not able to log out or to hide the icon yet, we are just letting it prepared for the future when the auth service is implemented. Modify *app.component.html* div tag as follows:

```
<div layout="row" layout-align="center center" td-toolbar-content flex>
  Jump The Queue
```

```

<span flex></span> //Fill empty space to put the icon in the right of the header
<button mat-icon-button mdTooltip="Log out"><mat-icon>exit_to_app</mat-icon></button>
</div>

```

If everything goes correctly, you should now have an icon at the right of the header no matter which view you are at.

Queue component

For our last view component we are going to use a component from Covalent Teradata: the **data table**. Let's begin.

As always: ng generate component queue-viewer and add a route in *app.routes.ts* to that component { path: 'queue', component: QueueViewerComponent},


Now we have the component created, let's take a bit of time to complete `navigateQueue()` function in `code-viewer` to point to this new component:

```

navigateQueue(): void {
  this.router.navigate(['queue']);
}

```

Back to our recently created component, it will be quite similar to the 2 others, but in this case, the body of the card will be a data table from covalent.

1. First, import the `CovalentDataTableModule` in `app.module.ts`: 


```

// Covalent imports
import {
  ...
  CovalentDataTableModule, // Add this line
} from '@covalent/core';

...

@NgModule({
  ...
  imports: [
    ...
    CovalentDataTableModule, // Add this line
  ],
  ...
})

```

2. Edit the HTML with the new table component: 

```

<td-layout>
  <mat-card>
    
    <mat-card-title>Queue view:</mat-card-title>

    <td-data-table
      [data]="queuers"
      [columns]="columns">
    </td-data-table>

    <div class="text-center pad-lg">
      <button mat-raised-button (click)="navigateCode()" color="primary" class="text-up">
    </div>

  </mat-card>
</td-layout>

```

Note

Learn more about Teradata data tables in OASP4Js [HERE](#)

What we did here is to create the component by its selector, and give the needed inputs to build the table: **columns** to display names and establish concordance with the data, and some **data** to show. Also, a button to return to the code view has been added following the same system as the navigation in code, but pointing to 'code':

```
export class QueueViewerComponent implements OnInit {
  columns: ITdDataTableColumn[] = [
    { name: 'code', label: 'Code' },
    { name: 'hour', label: 'Hour' },
    { name: 'name', label: 'Name' }];

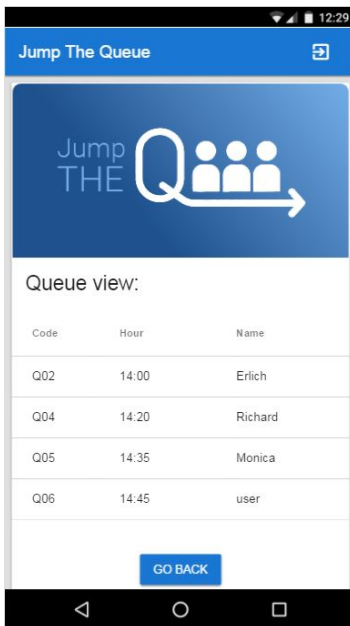
  queuers: any[] = [
    {code: 'Q04', hour: '14:30', name: 'Elrich'},
    {code: 'Q05', hour: '14:40', name: 'Richard'},
    {code: 'Q06', hour: '14:50', name: 'Gabin'},
  ];

  constructor(private router: Router) { }

  ngOnInit(): void {
  }

  navigateCode(): void {
    this.router.navigate(['code']);
  }
}
```

This will be the result:



Creating services


Note Learn more about services in OASP4Js [HERE](#)


At the moment we had developed all the basic structure and workflow of our application templates, but there is still some more work to do regarding security, calls to services and logic functionalities, this will be the objective of this second part of the tutorial. We will use angular/cli to generate our services as we did to create our components.

Note Learn more about creating new services in OASP4Js [HERE](#)

Auth service

We will start with the **security**, implementing the service that will store our state and username in the application, this services will have setters and getters of these two properties. This service will be useful to check when the user is logged or not, to show or hide certain elements of the headers and to tell the guard (service that we will do next) if the navigation is permitted or not.

To create the service we run: `ng generate service shared\authentication\auth` 

We navigate into this new service and we add this code as described above: 

```
import { Injectable } from '@angular/core';

@Injectable()
export class AuthService {
  private logged = false; // state of the user
  private user = ''; //username of the user

  public isLoggedIn(): boolean {
    return this.logged;
  }

  public setLogged(login: boolean): void {
    this.logged = login;
  }

  public getUser(): string {
    return this.user;
  }

  public setUser(username: string): void {
    this.user = username;
  }
}
```

When the access service will be done, it will call for this setters to set them with real information, and when we log off, this information will be removed accordingly.

As an example of use of this information service, we will move to *app.component.ts* and will add in the constructor the *AuthService* to inject it and have access to its methods.


Now on the template we are going to use and special property from Angular **ngIf** to show or hide the log-off depending on the state of the session of the user:


`<button *ngIf="auth.isLoggedIn()" mat-icon-button mdTooltip="Log out"><mat-icon>exit_to_app</mat-icon></button>`

This property will hide the log-off icon button when the user is not logged and show it when it is logged.


Note	Learn more about authentication in OASP4Js HERE
------	---

Guard service

With *AuthService* we have a service providing information about the state of the session, so we can now establish a guard checking if the user can pass or not trough the login page. We create it exactly the same way than the *AuthService*: `ng generate service shared\authentication\auth-guard` 

This service will be a bit different, because we have to implement an interface called *CanActivate*, which has a method called *canActivate* returning a boolean, this method will be called when navigating to a specified routes and depending on the return of this implemented method, the navigation will be done or rejected.

Note	Learn more about guards in OASP4Js HERE
------	---

The code should be as follows: 

```

import { Injectable } from '@angular/core';
import { CanActivate, Router } from '@angular/router';
import { AuthService } from './auth.service';

@Injectable()
export class AuthGuardService implements CanActivate {
  constructor(private authService: AuthService,
              private router: Router) {}

  canActivate(): boolean {

    if (this.authService.isLoggedIn()) { // if logged, return true and exit, allowing the navigation
      return true;
    }

    if (this.router.url === '/') {
      this.router.navigate(['access']); // if not logged, recheck the navigation to resend the request
    }

    return false; // and blocking the navigation.
  }
}

```

Now we have to add them to our *app.module.ts* providers array:

```

...
providers: [
  AuthGuardService,
  AuthService,
],
bootstrap: [AppComponent]
...

```

Finally, we have to specify what routes are secured by this guard, so we move to *app-routing.module.ts* and add the option "canActivate" to the paths to code-viewer and queue-viewer:

```


const appRoutes: Routes = [
  { path: 'access', component: AccessComponent},
  { path: 'code', component: CodeViewerComponent, canActivate: [AuthGuardService]},
  { path: 'queue', component: QueueViewerComponent, canActivate: [AuthGuardService]},
  { path: '**', redirectTo: '/access', pathMatch: 'full' }];

```

If you save all the changes, you will realize you can not go through access anymore, that is because we need to implement first our login function in the access service, which will change the value in AuthService and will let us navigate freely.

Access service

As we need to have this service in order to access again to our application, this will be the first service to be created. As always, ng generate service access/shared/access will do the job. Also remember to add the service to providers in app module.

This service will contain two functions, one for login when the button is pressed and other to log off when the icon button in the header is pressed. This functions will manage to set the values of the session and navigate properly. For now we are going to use a simple `if` to check if the user credentials are correct, in the future a server will do this for us. 

```

import { Injectable } from '@angular/core';
import { Router } from '@angular/router';
import { AuthService } from './auth.service';
import { MatSnackBar } from '@angular/material/snack-bar';

@Injectable()
export class AccessService {
  constructor(private auth: AuthService,
              public snackBar: MatSnackBar, // Angular Material snackbar component to show messages
              private router: Router) {}

  login(name, email, phone): void {
    if (name === 'user' && email === 'asd@asd.com' && phone === 123456789) { //check the credentials
      this.auth.setLogged(true); // if correct,
      this.auth.setUser(name);
    }
  }
}

```

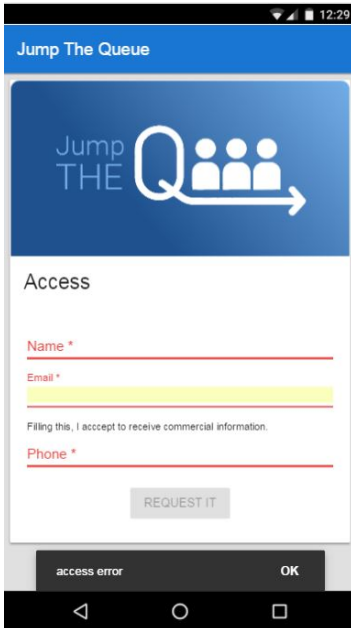



```

    this.router.navigate(['code']);
  } else {
    this.snackBar.open('access error', 'OK', {
      duration: 2000,
    });
  }
}

logoff(): void {
  this.auth.setLogged(false);
  this.auth.setUser('');
  this.router.navigate(['access']);
}
}
}

```



Now we have to inject this service in our AccessComponent in order to consume it. We inject the dependency into the component and we change our submit function to get the values from the form and to call the service instead of just always redirecting: 


```

export class AccessComponent implements OnInit {
  constructor(private accessService: AccessService) { }

  ngOnInit(): void {
  }

  submitAccess(formValue): void {
    this.accessService.login(formValue.value.name, formValue.value.email, formValue.value.phone,
    formValue.reset());
  }
}

```

This also has to be added to the template in order to pass the parameter into the function: 

```

<form layout="column" class="pad" (ngSubmit)="submitAccess(accessForm.form)" #accessForm

```

ngSubmit now passes as parameter the ngForm with the values introduced by the user.

Having this working should be enough to have again working our access component and grant access to the code and queue viewer if we introduce the correct credentials and if we do not, the error message would be shown and the navigation not permitted, staying still in the access view.

The last thing to do regarding security is to make functional our log-off icon button in the header, we move to `app.component.html` and add the correspondent (click) event calling for a function, in my case, called "logoff()".

```
<button *ngIf="auth.isLogged()" (click)="logoff()" mat-icon-button mdTooltip="Log out">
```

The name has to correspond with the one used in `app.component.ts`, where we inject `AccessService` so we can call its logoff function where the one from this components is called:


```
export class AppComponent {
  constructor(public auth: AuthService,
              private accessService: AccessService) {}

  logoff(): void {
    this.accessService.logoff();
  }
}
```

Once all of this is finished and saved, we should have all the workflow and navigation of the app working fine. Now it is time to receive the data of the application from a service in order to, in the future, call a server for this information.

Code Service

First step, as always, create the service in a shared folder inside the component: `ng generate service code-viewer/shared/code-viewer`.

Due to the simplicity of this view, the only purpose of this service is to provide the queue code, which will be generated by the server but, until we connect to it, we have to generate it in the service (*imports included here in order to make easier this section*): 

```
import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';

@Injectable()
export class CodeViewerService {

  constructor() { }

  getCode(): Observable<string> { // later, this will make a call to the server
    return of('Q06'); // but, for now, this Observable will do the work
  }
}
```

We return an Observable because when we implement calls to the server, we will use Http, and they return observables, so the best way to be prepared to this connection is having a simulation of the return of this Http calls.

It is time to inject it in the component and change a bit the variables to show in the template to get their value from auth and our code-viewer service:

```
export class CodeViewerComponent implements OnInit {

  code: string;
  name: string;

  constructor(private router: Router,
              private auth: AuthService,
              private codeService: CodeViewerService) { }
```

```

ngOnInit(): void {
  this.codeService.getCode().subscribe((data: string) => {
    this.code = data;
  });
  this.name = this.auth.getUser();
}

navigateQueue(): void {
  this.router.navigate(['queue']);
}
}

```

Note	Learn more about Observables and RxJs in OASP4Js HERE
------	---

Now if we log in the application, the name we introduce in the form will be the name displayed in the code-viewer view. And the queue code will be the one we set in the service.

Queue service

The last element to create in our application, as always: `ng generate service queue-viewer/shared/queue-viewer` and then add the service in providers at `app.module.ts`.

This service will work the same way code-viewer, it will simulate an observable that returns the data that will be displayed in the data table of Covalent Teradata:

```

import { Observable } from 'rxjs';

export class QueueViewerService {
  queuers: any[];

  constructor() { }

  getQueuers(): Observable<any[]> { // later, this will make a call to the server and

    this.queuers = [{ code: 'Q04', hour: '14:30', name: 'Elrich' },
      { code: 'Q05', hour: '14:40', name: 'Richard' },
      { code: 'Q06', hour: '14:50', name: 'Gabin' }];

    return of(this.queuers); // but, for now, this Observable will do the work
  }
}

```

And the `queue-viewer.component.ts` will be modified the same way:

```

export class QueueViewerComponent implements OnInit {

  columns: ITdDataTableColumn[] = [
    { name: 'code', label: 'Code'},
    { name: 'hour', label: 'Hour' },
    { name: 'name', label: 'Name'}];

  queuers: any[];

  constructor(private router: Router,
    private queueService: QueueViewerService) { }

  ngOnInit(): void {
    this.queueService.getQueuers().subscribe( (data) => {
      this.queuers = data;
    });
  }

  navigateCode(): void {
    this.router.navigate(['code']);
  }
}

```

At the moment, we have a functional application working exclusively with mock data, but we want to connect to a real back-end server to make calls and consume its services to have more realistic data, the way we implemented our components are completely adapted to read mock data or real server data, that is why we use services, to isolate the origin of the logic and the data from the component. Is the code of our services what is going to change, and we will go to see it now.

Making calls to server

At this point we are going to assume you have finished the OASP4J JumpTheQueue tutorial or, at least, you have downloaded the project and **have it running locally on localhost:8081**.

With a real server running and prepared to receive calls from our services, we are going to modify a bit more our application in order to adjust to this new status.

First, some configurations and modifications must be done to synchronize with how the server works:



1. Now our *Authentication.ts* should have the parameter "code" along with its getters and setters, which will be the queue code of the user, this has been moved here because this information comes from the register call when we access, not when we load the code view.
2. Completely remove shared service from *code-viewer* folder, because, at this moment, the only purpose of that folder was to store a service which loads the queue-code of the user, as it is not used anymore, this service has no sense and the *code-viewer.component* now loads its code variable from *auth.getCode()* function.
3. Create a file called *config.ts* in *app* folder, this config will store useful global information, in our case, the *basePath* to the server, so we can have it in one place and access it from everywhere, and even better, if the url changes, we only need to change it here:

```
export const config: any = {
  basePath: 'http://localhost:8081/jumpthequeue/services/rest/',
};
```

Once done all the preparations, let's move to *acces.service.ts*, here we had a simple *if* to check if the user inputs are what we expected, now we are going to call the server and it will manage all this logic to finally return us the information we need.

To call the server to are going to import *Angular HttpClient* class from *@angular/common/http*, this class is the standard used by angular to make Http calls, so we are going to use it. The register call demands 3 objects: name, email and phone, so we are going to build a post call and send that information to the proper URL of that server service, it will return an observable and we have already worked with them: first we map the result and then we subscribe to have all the response data available, also we implement the error function in case something went wrong. The new register function should be as follows:

```
register(name, email, phone): void {
  this.http.post<any>(`${config.basePath}visitormanagement/v1/register`, {name: name, email: email, phone: phone})
    .subscribe( (res) => {
      this.auth.setLogged(true);
      this.auth.setUser(name);
      this.auth.setCode(res.code.code);
      this.router.navigate(['code']);
    }, (err) => {
      this.snackBar.open(err.error.message, 'OK', {
        duration: 5000,
      });
    });
}
```

Important: As we can see in the code the request is mapped with the type *any*. This is made for this tutorial purposes, but in a real scenario this *any* should be changed by the correct type (interface or class) that fits with the Http response.

As we can see, and mentioned before, our preparations to this server call we have done previously let us avoid changing anything in access component or template, everything should be working only doing that changes.

Our queue-viewer will need some modifications as well, in this case, both component and services will be slightly modified. *queue_viewer.service* will make a call to the server services as we done in *access.service* but in this case we are not going to implement a subscription, that will be components task. So `getQueuers()` should look like this:

```
getQueuers(): Observable<any> {
  return this.http.post<any>(`${config.basePath}visitorsmanagement/v1/visitor/search`, {})
  // the post usually demands some parameters to paginate or make
  // in this case we do not need nothing to do more
}
```

Regarding *queue-viewer.component* we need to modify the columns to fit with the data received from the server and the template will be modified to use **async pipe** to subscribe the data directly and a loader to show meanwhile.

About the columns, the server sends us the data array composed of two objects: *visitor* with the queue member information and *code* with all the code information. As we are using the name of the queuer, the time it is expected to enter and its code, the column code should be like this:

```
columns: ITdDataTableColumn[] = [
  { name: 'visitor.name', label: 'Name'},
  { name: 'code.dateAndTime', label: 'Hour', format: ( (v: string) => moment(v).format('LL')
  { name: 'code.code', label: 'Code'},
];
```

Additionally, server sends us the date and time as timestamp, so we need to use **moment.js** to format that data to something readable, to make that, just use the format property from Teradata Covalent columns.

Finally, to adapt to async pipe, `ngOnInit()` now does not subscribe, in its place, we equal the `queuers` variable directly to the `Observable` so we can load it using the `*ngIf - else` structure to show the loading bar from Material and load the queuers in the template:

```
<td-layout>
  <mat-card *ngIf="queuers | async as queuersList; else loading" > // load queuers and assign
    
    <mat-card-title>Queue view:</mat-card-title>

    <td-data-table
      [data]="queuersList.result"
      [columns]="columns">
      <ng-template tdDataTableTemplate="visitor.name" let-value="value" let-row="row" let-cc
        <div layout="row">
          <span *ngIf="value === auth.getUser(); else normal" flex><b>{{value}}</b></span>
          <ng-template #normal>
            <span flex>{{value}}</span>
          </ng-template>
        </div>
      </ng-template>
    </td-data-table>

    <div class="text-center pad-lg">
      <button mat-raised-button (click)="navigateCode()" color="primary" class="text-upper">
    </div>

  </mat-card>

  <ng-template #loading> // template to show when the async pipe is loading data
    <mat-progress-bar
      color="accent"
      mode="indeterminate">
    </mat-progress-bar>
```

```
</ng-template>
```

```
</td-layout>
```

Also, to make easier to the user read what is his position, Covalent Teradata provides with a functionality to check columns and modify the value shown, we used that to make bold the name of the user which corresponds to the user who is registered at the moment.

That is all regarding how to build your own OASP4Js application example, now is up to you add features, change styles and do everything you could imagine. Just one final step to complete the tutorial, run the tutorial outside your local machine: Deployment.

Next chapter: Deploy your OASP4Js app
