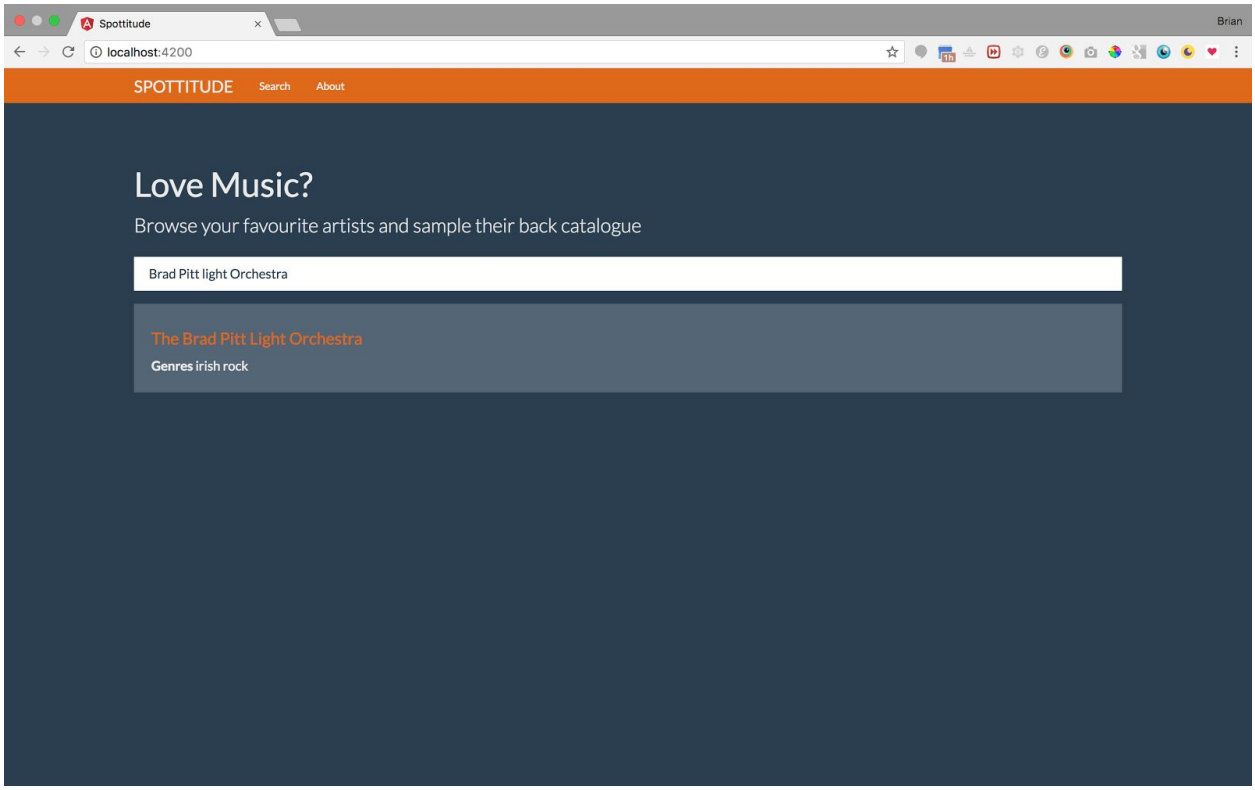
# Project Spottitude




## Goal

In this project we'll use the Spotify API to search artists, display their their back catalogue and  preview album tracks

# Steps

# Part 1 - Setting up the core navigational elements

1. Create a new Angular 2 project called **spottitude**
    a. `ng new spottitude`
    b. **ng serve** to test installation
    c. Verify by open the project in a browser using "http://localhost/4200"

2. In your terminal/Git bash/powershell navigate to the **app** directory of your project and create a new directory called **components** using the **mkdir** command. As we build on our project we'll add our 5 project components to this directory

4. Now let's create our first 3 components. In your terminal/Git bash/powershell navigate to the **components** directory of your project and create a 3 new components:
    a. `ng g component about`
    b. `ng g component search`
    c. `ng g component navbar`
5.

6. We'll use bootstrap to style the project. So let's grab a pre-built **navbar** from **https://getbootstrap.com/**
    a. Go to getting started -> examples -> starter template
    b. Clink on template to view
    c. View source
    d. Copy the nav section
    e. Paste it into our **navbar.component** helper html file Modify the code to look like the following. Notice that the standard anchor tag **href** attributes have been replaced with **routerLink** attributes. We'll use

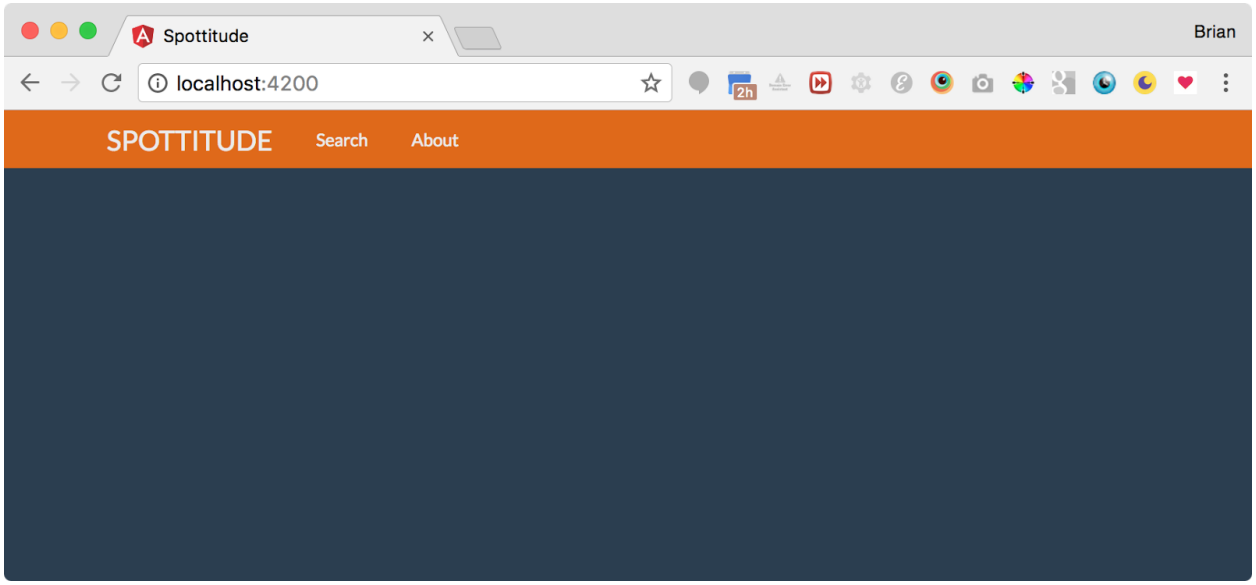
these for our Angular navigation routing. Remember they are used navigate based on a pattern match on a requesting URL

```
<nav class="navbar navbar-inverse">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle collapsed"
data-toggle="collapse" data-target="#navbar" aria-expanded="false"
aria-controls="navbar">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" routerLink='/'>SPOTTITUDE</a>
        </div>
        <div id="navbar" class="collapse navbar-collapse">
            <ul class="nav navbar-nav">
                <li><a routerLink='/'>Search</a></li>
                <li><a routerLink='/about'>About</a></li>
            </ul>
        </div>
        <!--/.nav-collapse -->
    </div>
</nav>
```

7. Include a reference to the navbar in **app.component.html** so it is displayed
    a. `Add <app-navbar></app-navbar>` to the html

8. Now let's go and get a basic bootstrap theme to give us a head start on styling our application.

a. Go to [bootswatch.com](bootswatch.com) and choose a theme you like (superhero is pretty nifty but that's just one opinion...)
b. Click **download**
c. Copy the url from the address bar
d. Create a CSS link in the head section of **index.html** (our main html template).
e. Paste the bootswatch theme URL into the **href** attribute.
f. Save and check your results in the browser. It should look like below.



9. Now we'll wire up the routing from our navbar links to the **search** and **about components.**
    a. Manually create a new route file within the **app** directory
        i. Save it as **app.routing.ts**
        ii. Import the **search** and **about** components and wire up the routes as per below (the **search** will be our default route)

```
import {ModuleWithProviders}from '@angular/core';
import {Routes, RouterModule} from '@angular/router';

import {AboutComponent} from './components/about/about.component';
import {SearchComponent} from './components/search/search.component';

const appRoutes: Routes = [
      {
            path:'',
            component:SearchComponent
      }
      ,
      {
          path:'about',
            component:AboutComponent

      }
];

export const routing:ModuleWithProviders = RouterModule.forRoot(appRoutes);
```

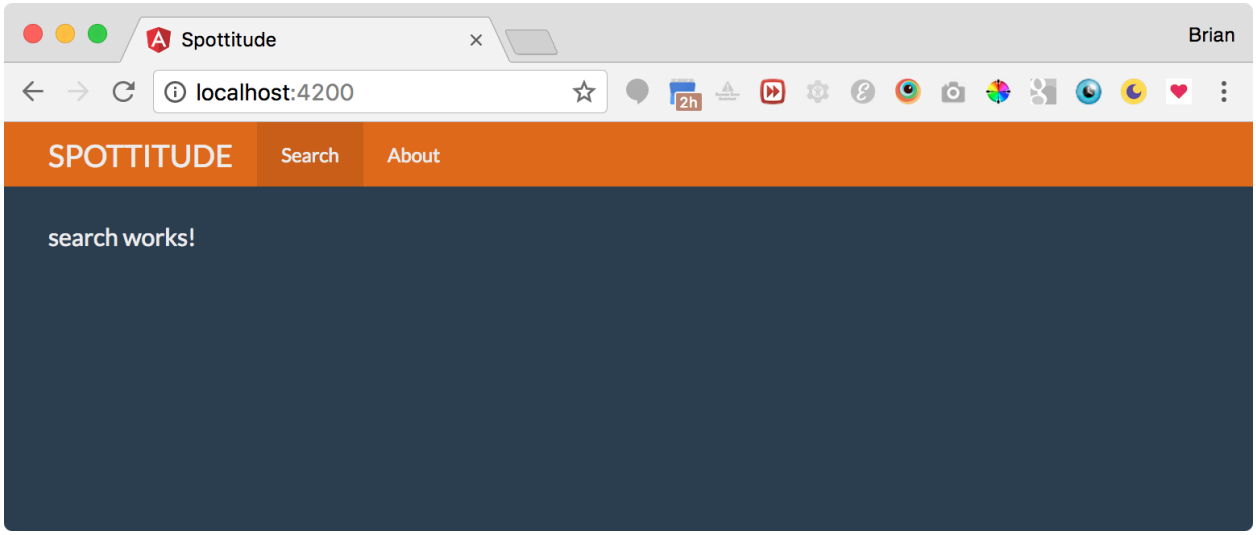
9. We then need a `<router-outlet>` placeholder to inject our route destination content. To do this, add the code below to **app.component** below the **<app-navbar>** element. (We're also dressing it up with some bootstrap and custom styling)

```
<div class="main">
      <div class="container">
      <router-outlet></router-outlet>
      </div>
</div>
```

10. We need to do one more thing before the routing works.
    a. Go to **app.module.ts** and register the **routing** file.
        i. Import the file
            - `import { routing } from './app.routing';`
        ii. Register the routes

```
imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    routing
],
```

11. Save and check your result in the browser. You should see the search component displayed. (Remember, it's the default route )



# Part 2 - Setting up the core API call related elements

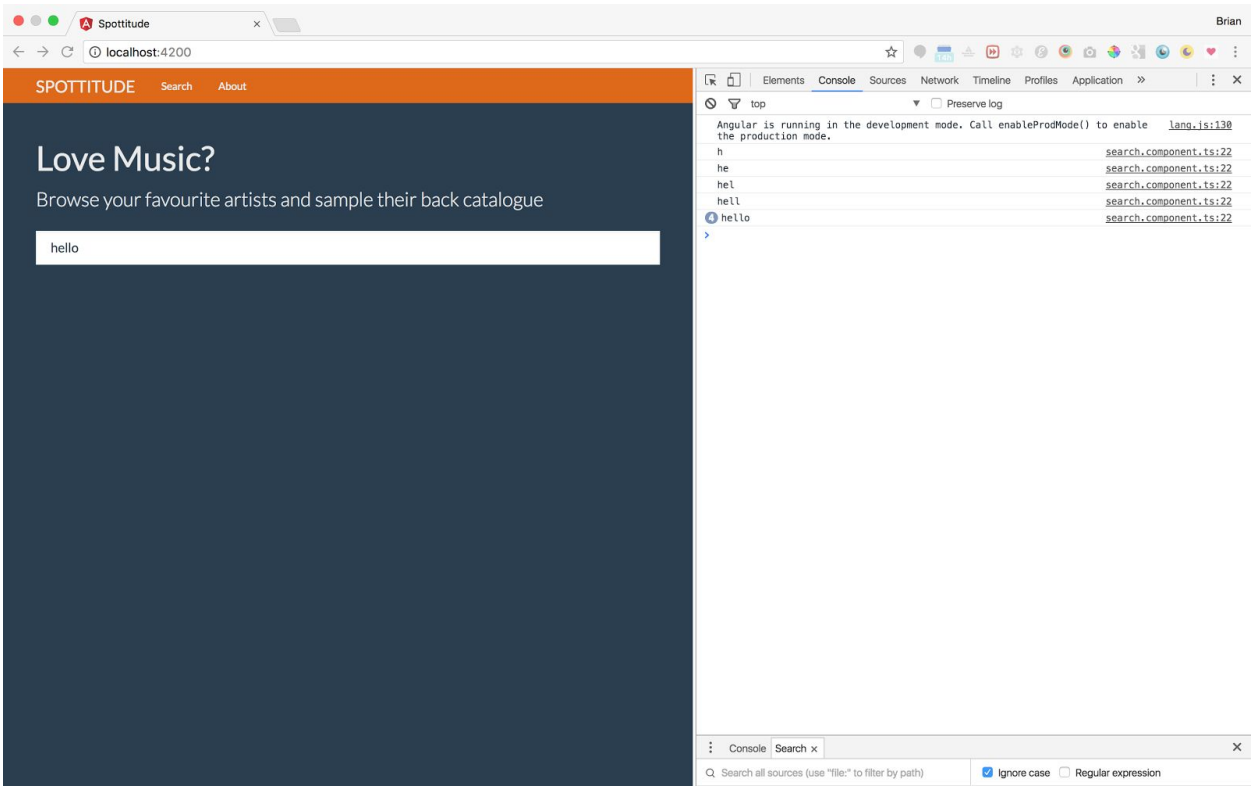
12. In your **search** component, add a property called searchStr of type string. We'll use this to bind out data from an input form on the **search.component.html** helper file. Then add method called **searchMusic()** that will write our search data to the console as a test. This method will later call the Spotify service . The code should look like below.

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-search',
  templateUrl: './search.component.html',
  styleUrls: ['./search.component.css']
})
export class SearchComponent implements OnInit {

  private searchStr:string;

  constructor() {
  }

  ngOnInit() {
  }

  searchMusic(){
      console.log(this.searchStr);
  }

}
```

13. Update the **search.component.html** helper file to include some explanation text and  a form with a single text field used for searching . The **ngModel** will bind any input to the **searchStr** that we just created in the search component. The **keyup** event will trigger a call to the **searchMusic()** method

```
<h1>Love Music?</h1>
<p class="lead"> Browse your favourite artists and sample their back
catalogue</p>
<form>
  <div class="form-group">
  <input type="text" name="searchStr" class="form-control"
placeholder="Search Artists...." [(ngModel)]="searchStr"
(keyup)="searchMusic()">
    </div>
</form>
```

Test your results in dev tools




14. In your terminal/Git bash/powershell navigate to the **app** directory of your project and create a new directory called **services** using the **mkdir** command. In here we'll create a service that will talk to the Spotify API. The service is directly responsible for accessing the Spotify API. For now, our **search** component will use the service ("has a" relationship) to fetch the data. The search will bind the returned service data to its HTML template for viewing in the browser.
    **b. `ng g service music`**
    (Ignore the warning)
11. Open the newly created **music.service.ts** file (it won't be created in its own directory)
12. Manually add the following imports below the existing ***Injectable*** import at the top of the file. These will be used to:
    a. Use HTTP to get the data from the API

b. Map the returned data to JSON format for us to use in our project using an Observable . (Our code will react when streaming data is available)

```
// manually import these
import { Http, Headers} from '@angular/http';
import 'rxjs/add/operator/map';
```

13. Run your project to check that everything is still OK

14. Create a **searchUrl** property inside the service that we'll uses to access the Spotify API.

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

// manually import these
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';

@Injectable()
export class MusicService {
  private searchUrl: string;
  constructor() {
  }
}
```

13. Update the service constructor
    a. Add a parameter of type HTTP to the constructor

i. *(We have leveraged TypeScript's constructor syntax for declaring parameters and properties simultaneously.)*

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

// manually import
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';

@Injectable()
export class MusicService {

  private searchUrl: string;

  constructor(private _http:Http) {

  }
}
```

14. Add a **searchMusic()** method to your service. This will fetch your spotify artist details. This is the actual call to the Spotify API. It's just a URL with
    a. An instruction to get a list of artists that match a particular search string:

    This method will use a query string to get your artists info. This then maps the returned info into JSON format for us to use. This method will later be called from within our **search** component

```
searchMusic(str:string, type='artist'){
this.searchUrl = "https://api.spotify.com/v1/search?query="
                        +str+"&offset=0&limit=20&type="
                        +type+"&market=US";

return this._http.get(this.searchUrl).map(res => res.json());
```

```
}
```

15. Import our service class into our **search** component for use *(note the **../../** at the start of the path)*

```
// manually import this service
import { MusicService } from '../../services/music.service';
```

16. We also need to import the service into **app.module.ts** in order to register it as as a service provider

```
import { MusicService } from './services/music.service';

...

providers: [MusicService],
bootstrap: [AppComponent]
```

17. Now let's update the **search** constructor to use the service.

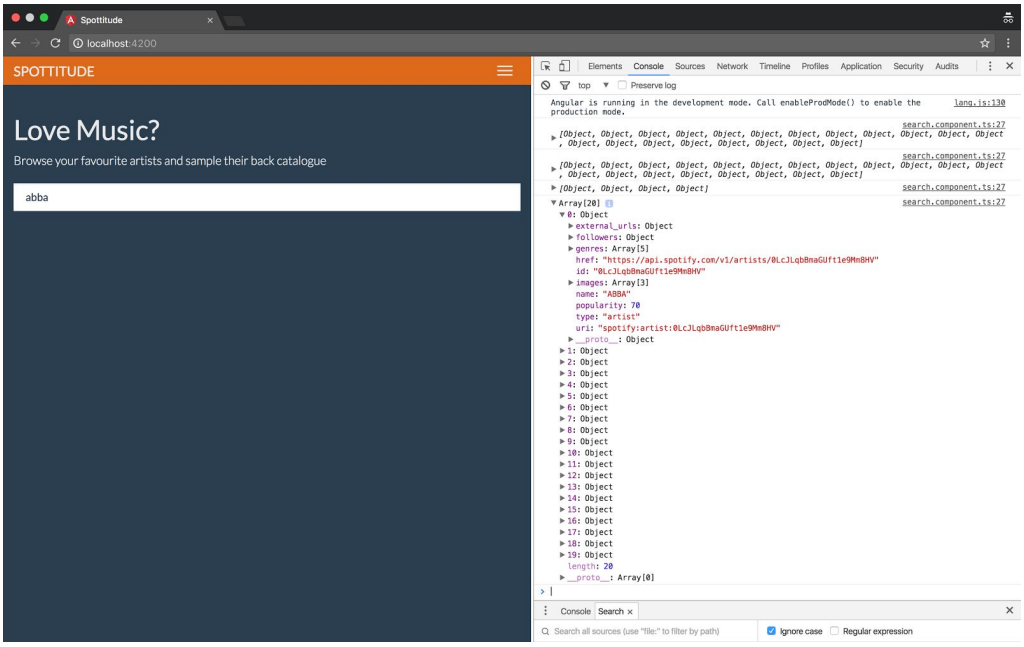
```
// inject MusicService as a dependency
constructor(private _musicService:MusicService) {
}
```

18. The modify the **searchMusic()** function in **search** to call the corresponding method in the music service.

```
searchMusic(){
//console.log(this.searchStr);
this._musicService
      .searchMusic(this.searchStr)
                        .subscribe(res =>
                              {console.log(res.artists.items)}
                        );
```

}

19. Open up chrome dev tools to see if the service correctly fetched artist info. Once you type in an artist name you should see a list of objects appear in the console. Your main call to the API is now working. Good job!



# Part 3 - Displaying artist info on screen

20. We are next going to create 2 model classes that we'll use to display our artist related data. These are:
    a. Artist
    b. Album

    To do this we'll create our files *outside* the **app** directory
    1. Create a new file called **album.ts**
        a. Add the following code to the file:

```
export class Album{
        id:number;
}
```

    2. Create a second file called artist.ts
        a. Add the following code

```
import {Album} from './album'
export class Artist
{
        id:number;
        name:string;
        genres;any;
        albums:Album[];
}
```

    These model classes will allow us to create arrays of artists and their albums in order to display them on screen.

21. Now that we have models in place let's use the **Artist** class in our **search** component
    a. Import the **Artist** class into the search component

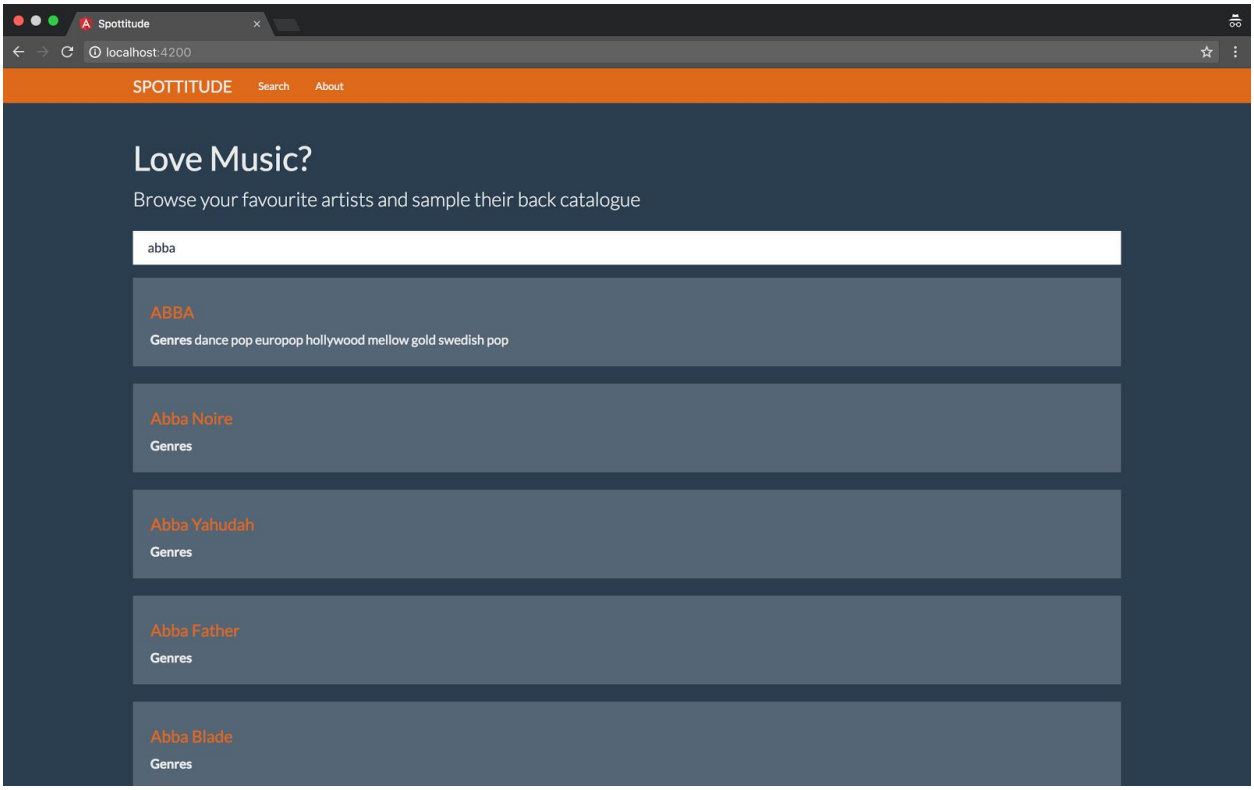
b. Create a new property called **searchRes: Artist[]** that we'll use to store our search results. We'll also use this property to display the results to the screen. Modify the search component as in bold below.

```
import { Component, OnInit } from '@angular/core';
import { MusicService } from '../../services/music.service';
import {Artist} from '../../../artist';

@Component({
  selector: 'app-search',
  templateUrl: './search.component.html',
  styleUrls: ['./search.component.css']
})
export class SearchComponent implements OnInit {

  private searchStr:string;
  private searchRes:Artist[];

  constructor(private _musicService:MusicService) {
  }

  ngOnInit() {
  }

  searchMusic(){
      //console.log(this.searchStr);
      this._musicService
      .searchMusic(this.searchStr)
      .subscribe(res => {this.searchRes = res.artists.items)});
  }
}
```

22. Modify your **search.component.html** to display the search results
    a. Add the html below the form
    b. A list of artists will only be displayed if there is a search result
    c. A list of genres for each artist will also be displayed
    d. Notice that we have added a link that when clicked will display any **albums** associated with an artist **id.** The **routerLink** passes this **id** as part of its redirect

```
<div *ngIf="searchRes">
    <div *ngFor="let artist of searchRes">
        <div class='row'>
            <div class="col-md-12">
                <div class="search-res well">
                    <h4><a
routerLink="/artist/{{artist.id}}">{{artist.name}}</a></h4>
                    <div>
                        <strong>Genres</strong>
                        <span *ngFor="let genre of artist.genres">
                              {{genre}} 
                        </span>

                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
```

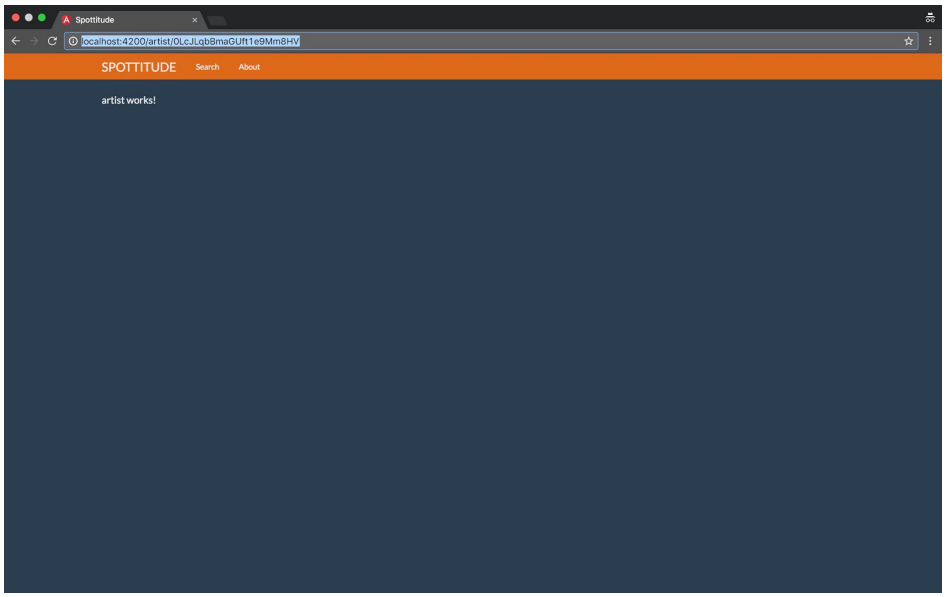
Check the browser to see the results of a search.




23. Ok. We're looking good. Now let's wire up the links from artists to their album back catalogue.
    a. First we create a new **artist** component in the **components** directory
        i. **ng g component artist**
    b. Then we update our **routing** file to wire up clicks on the artist name that will take the user to the artist detail view
        i. Add the new route as per below (don't forget the comma...)
        ii. Notice that this route path looks for an artist id. This id will be used to make another call to the Spotify API to get artist details

```
,
{
    path:'artist/:id',
      component:ArtistComponent

}
```

24. Click on an artist that was returned from a search . You should see the barebones artist component view. Notice the artist **ID** in the browser URL field



25. Modify the **artist.component.ts** file to call  to get an artist's information and their albums.
    a. To get the code below working we'll next need to create corresponding **getArtist()** and **getAlbums()** methods in the **music** service.
    b. We want to make a call to the Spotify API each time this component is injected into the view. We also want to capture the artist id in order to get that artist's details and related album catalog. To do this we make our calls to the API in our **OnInit()** method. **OnInit()** is the preferred method to use when binding data from a view/template.
    c. The ActivatedRoute refers the current route being used. In this case the route is activated from **search.component.html**
        i. `routerLink="/artist/{{artist.id}}`
    d. By subscribing to the API methods, Angular can react to changes in the data and bind the results to the view

```
import { Component, OnInit } from '@angular/core';
import { Router, ActivatedRoute } from '@angular/router';
import { MusicService } from '../../services/music.service';
import { Artist } from '../../../artist';
import { Album } from '../../../album';

@Component({
  selector: 'app-artist',
  templateUrl: './artist.component.html',
  styleUrls: ['./artist.component.css']
})
export class ArtistComponent implements OnInit
{

      id:string;
      artist:Artist[];
      albums:Album[];

  constructor(private _musicService:MusicService,
```

```
        private _route:ActivatedRoute ) {}

    ngOnInit(){
         this._route.params
             .map(params => params['id'])
             .subscribe((id) => {
                 this._musicService.getArtist(id)
                     .subscribe(artist => {
                         this.artist = artist;
                     })
                 this._musicService.getAlbums(id)
                     .subscribe(albums => {
                         this.albums = albums.items;
                     })
             });
     }
}
```

26. Modify the **music** service to include the method calls to the Spotify API. See the changes in bold

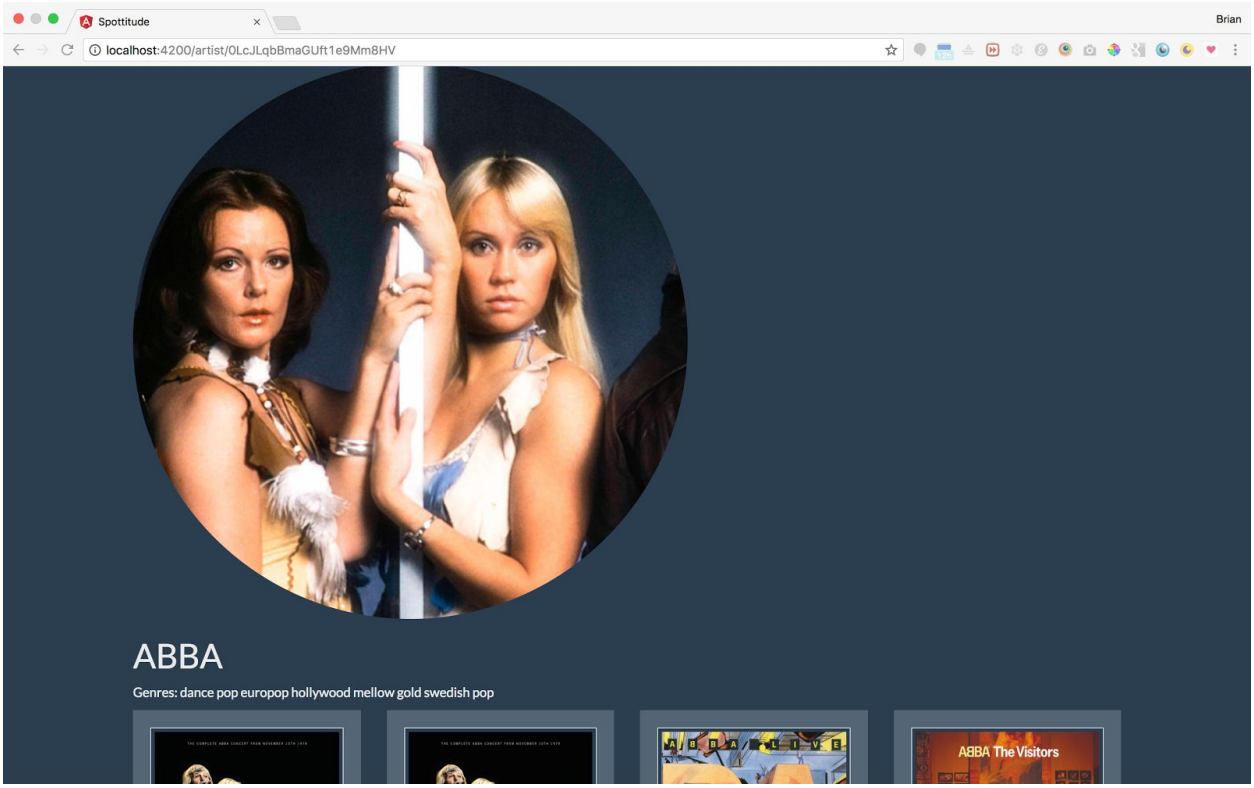
```
import { Injectable } from '@angular/core';
// manually import
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';

@Injectable()
export class MusicService {

  private searchUrl: string;
  private artistUrl: string;
  private albumsUrl: string;

  constructor(private _http:Http) {

  }

  searchMusic(str:string, type='artist'){
      this.searchUrl = "https://api.spotify.com/v1/search?query="
                              +str+"&offset=0&limit=20&type="
                              +type+"&market=US";

      return this._http.get(this.searchUrl).map(res => res.json());
  }
 getArtist(id:string)
  {
      this.artistUrl = "https://api.spotify.com/v1/artists/"+id;
      return this._http.get(this.artistUrl).map(res => res.json());
  }

  getAlbums(artistId:string)
  {
      this.albumsUrl =
"https://api.spotify.com/v1/artists/"+artistId+"/albums";
      return this._http.get(this.albumsUrl).map(res => res.json());
```

```
  }
}
```

27. Bind the returned data to the **artist.component.html** using the code below

```
<div *ngIf="artist">
    <header>
     <div *ngIf="artist.images.length > 0">
      <img class="artist-thumb img-circle" src="{{artist.images[0].url}}">
      </div>
        <h1>{{artist.name}}</h1>
        <p *ngIf="artist.genres.length > 0">
            Genres: <span *ngFor="let genre of
artist.genres">{{genre}} </span>
        </p>
    </header>
    <div class="artist-albums">
        <div class="row">
            <div *ngFor="let album of albums">
                <div class="col-md-3">
                    <div class="album well">
                        <img class="album-thumb img-thumbnail"
src="{{album.images[0].url}}">
                        <h4>{{album.name}}</h4>
                        <a class="btn btn-default btn-block"
routerLink="/album/{{album.id}}">Album Details</a>
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
```

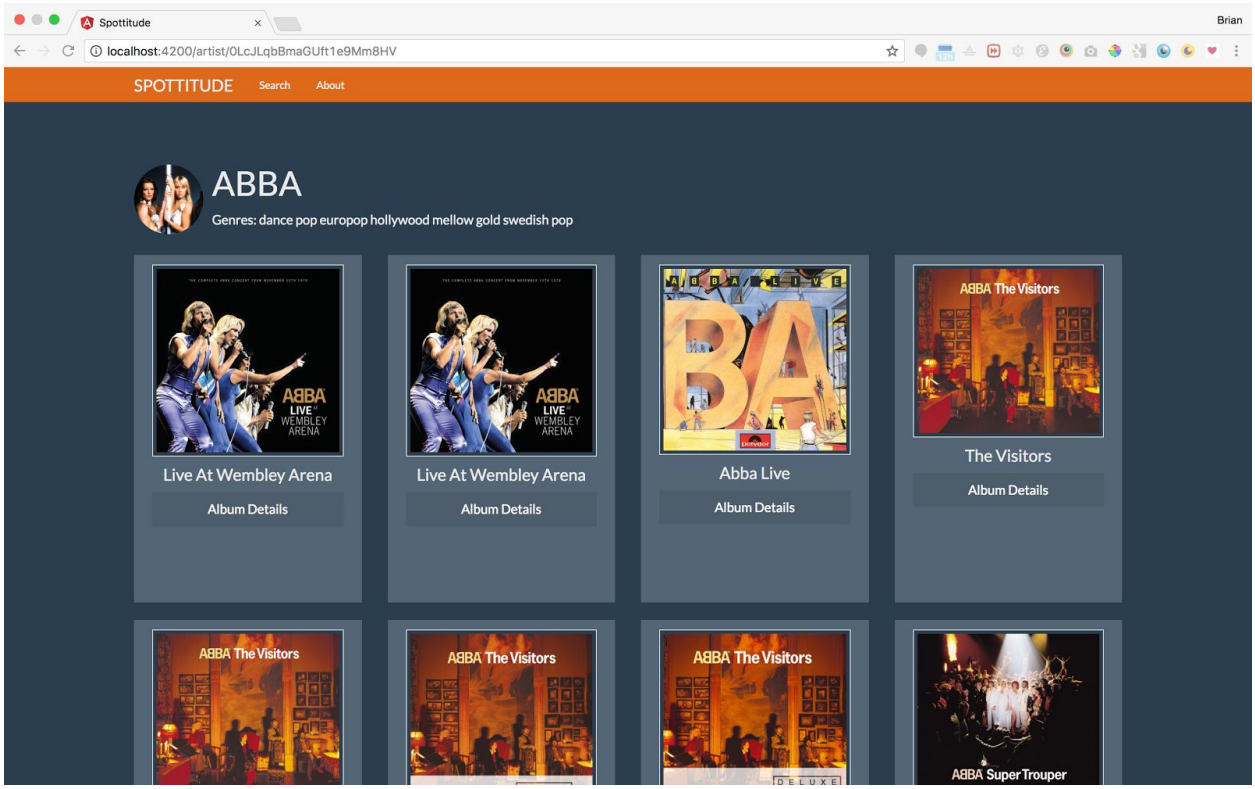
View your changes in the browser



28. The styling looks off. So let's fix that.
    a. Go to the main styles.css file and add the following code

```
.main {
    padding: 30px;
}

.artist-thumb {
    width: 80px;
    height: auto;
    float: left;
    margin-right: 10px;
}

.artist-header {
    padding-bottom:20px;
    margin-bottom: 20px;
 }

.artist-albums .well{
      margin-bottom: 20px;
      overflow: auto;
      min-height: 400px
}

.album {
      text-align: center;
      padding: 10px 20px;
}

.album-thumb{
      width: 100%;
}

header{
      padding-bottom: 20px;
}
```

View your changes in the browser: Much better



# Part 4 - Preview album tracks

29. In our final part we will display album tracks when a user clicks on an album listed in the **artist** component. To do this we need to
    a. Update the **routing** to redirect to the album tracks
    b. Create an **album** component that will display the tracks.
30. Create a new route. Add the code below to your existing routes.
    a. Notice that, like the artist route, the album route uses the **album id** to search for and return album tracks. This route originated in the **artist.component.html** view
        i. `routerLink="/album/{{album.id}}`

```
,
{
    path:'album/:id',
      component:AlbumComponent
}
```

31. This won't work yet becayse we haven't yet created the **AlbumComponent** Let's do that now.
    a. create a new **album** component in the **components** directory
        i. ng g component album
    b. then import the component into the **app.routing.ts** file for use
    c. Modify the album component to call the Spotify API. This time we need a **getAlbum()** method get the tracks for a particular album

```
import { Component, OnInit } from '@angular/core';
import { Router, ActivatedRoute } from '@angular/router';
import { MusicService } from '../../services/music.service';
import { Artist } from '../../../artist';
import { Album } from '../../../album';

@Component({
  selector: 'app-album',
  templateUrl: './album.component.html',
  styleUrls: ['./album.component.css']
})
export class AlbumComponent implements OnInit {

    id:string;
    album:Album[];

  constructor(private _musicService:MusicService,
      private _route:ActivatedRoute ){

  }

  ngOnInit(){
        this._route.params
            .map(params => params['id'])
            .subscribe((id) => {
                this._musicService.getAlbum(id)
                    .subscribe(album => {
                        this.album = album;
                    })
            });
    }

}
```

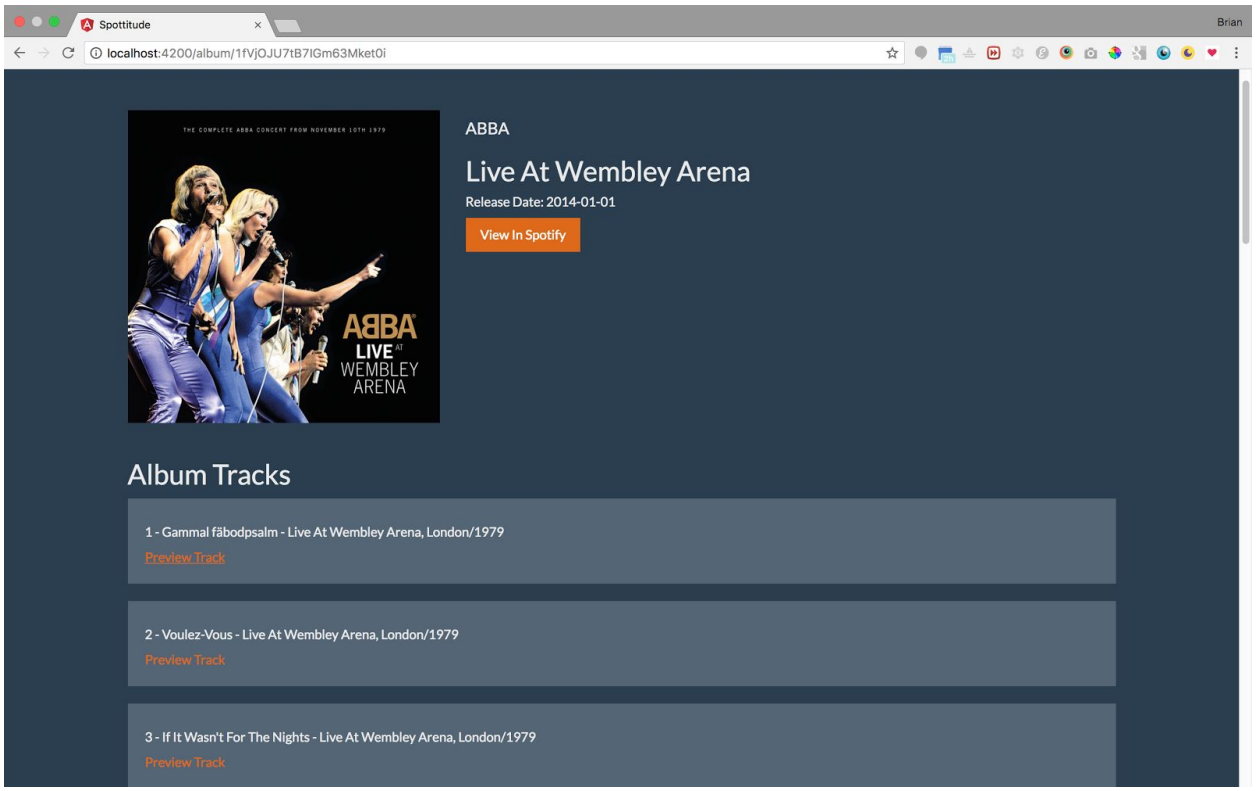
32. For this to work we the need to add the corresponding **getAlbum()** method and an **`albumUrl`** property in our **music** service

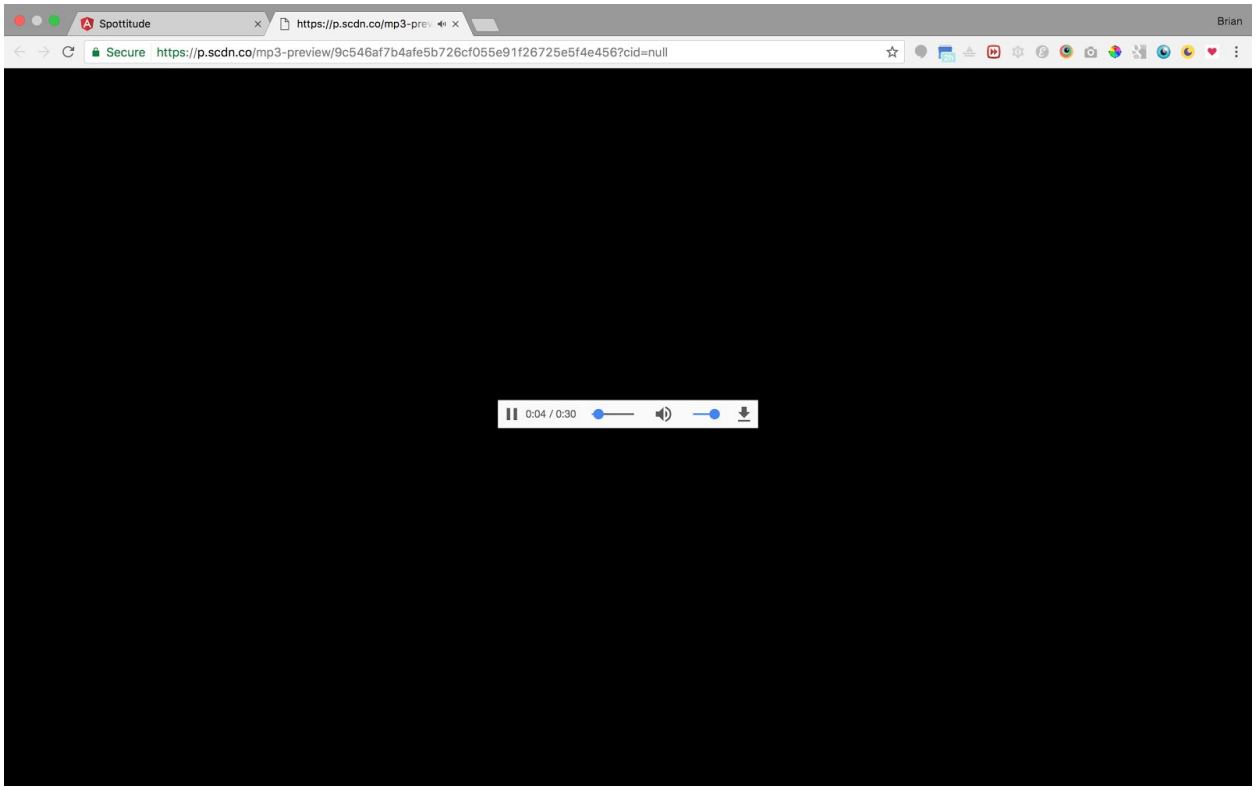
```
getAlbum(id:string)
  {
      this.albumUrl = "https://api.spotify.com/v1/albums/"+id;
      return this._http.get(this.albumUrl).map(res =>
res.json());
  }
```

33. Add the following code to **album.component.html**

```
<div id="album" *ngIf="album">
    <header class="album-header">
        <div class="row">
            <div class="col-md-4">
                <img class="album-thumb" src="{{album.images[0].url}}">
            </div>
            <div class="col-md-8">
                <h4 *ngIf="album.artists.length > 0">
                <span *ngFor="let artist of
album.artists">{{artist.name}}</span></h4>
                <h2>{{album.name}}</h2>
                <h5>Release Date: {{album.release_date}}</h5>
                <a class="btn btn-primary" target="_blank"
href="{{album.external_urls.spotify}}">View In Spotify</a>
            </div>
        </div>
    </header>
    <div class="album-tracks">
        <h2>Album Tracks</h2>
        <div *ngFor="let track of album.tracks.items">
            <div class="well">
                <h5>{{track.track_number}} - {{track.name}}</h5>
                <a href="{{track.preview_url}}" target="_blank">Preview
Track</a>
            </div>
        </div>
    </div>
</div>
```

View the results in the browser: And we're  done!




Spottitude
https://p.scdn.co/mp3-prev
Brian
Secure https://p.scdn.co/mp3-preview/9c546af7b4afe5b726cf055e91f26725e5f4e456?cid=null
0:04 / 0:30