ABOUT MATT RAIBLE

Java Champion and Developer Advocate @okta with a passion for skiing, mtn biking, VWs, & good beer.

Angular 8 + Spring Boot 2.2: Build a CRUD App Today!

Posted by: Matt Raible in Enterprise Java May 27th, 2019 2 Comments 1358 Views

"I love writing authentication and authorization code." ~ No Java Developer Ever. Tired of building the same login screens over and over? Try the Okta API for hosted authentication, authorization, and multi-factor auth.

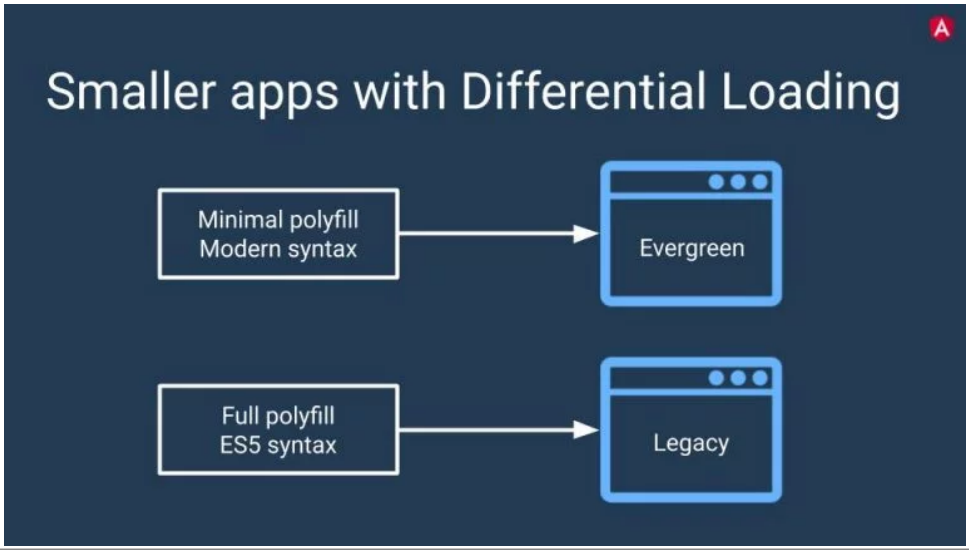
If you've been a Java developer for more than 15 years, you probably remember when there were a plethora of Java web frameworks. It started with Struts and WebWork. Then Tapestry, Wicket, and JSF came along and championed the idea of component-based frameworks. Spring MVC was released in 2004 (in the same month as Flex 1.0 and JSF 1.0) and became the de-facto standard in Java web frameworks over the next six years.

Then along came AngularJS and everyone started moving their UI architectures to JavaScript. Angular 2 was announced at the same time that Spring Boot was first revealed in 2014, and it took a couple of years for it to be released, solidify, and become a viable option. These days, we call it Angular, with no version number. The last few releases have been pretty darn stable, with smooth upgrade paths between major releases.

Today, I'd like to show you how to build an app with the latest and greatest versions of Angular and Spring Boot. Angular 8 and Spring Boot 2.2 both come with performance improvements to make your developer life better.

What's New in Angular 8?

Angular 8 adds differential loading, an optional Ivy Renderer, and Bazel as a build option. Differential loading is where the CLI builds two separate bundles as part of your deployed application. The modern bundle is served to evergreen browsers, while the legacy bundle contains all the necessary polyfills for older browsers.



What's New in Spring Boot 2.2?

Spring Boot, feeling some heat from quick-starting frameworks like Micronaut and Quarkus, has made many performance improvements as well. JMX is now disabled by default, Hibernate's entity scanning is disabled, and lazy initialization of beans is on by default. In addition, startup time and memory usage have been reduced by making use of

```
proxyBeanMethods=false
```

in Spring Boot's

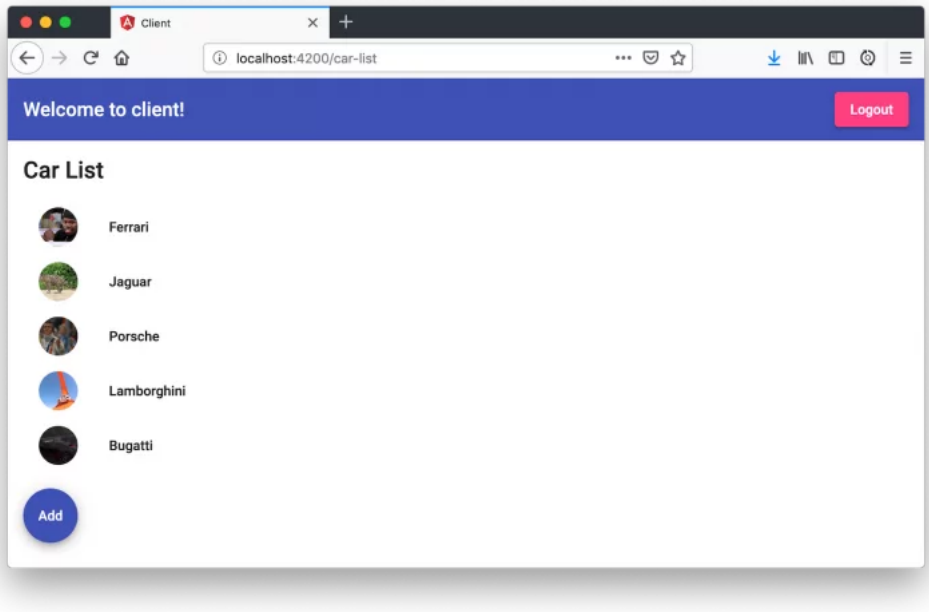
```
@Configuration
```

classes. See Spring Boot 2.2 Release Notes for more information.

If you're stuck on older versions of these frameworks, you might want to check out a couple of my previous posts:

- Build a Basic CRUD App with Angular 7.0 and Spring Boot 2.1
- Build a Basic CRUD App with Angular 5.0 and Spring Boot 2.0

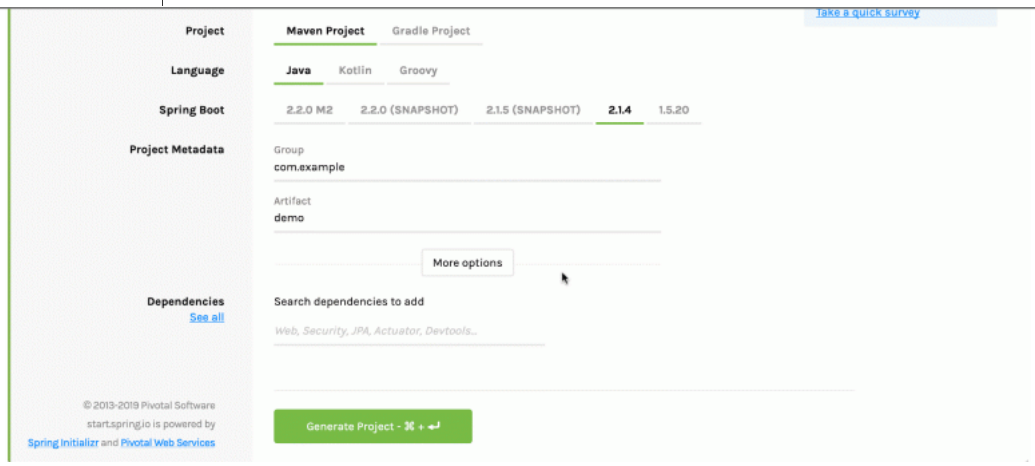
This post describes how to build a simple CRUD application that displays a list of cool cars. It'll allow you to edit the cars, and it'll show an animated gif from GIPHY that matches the car's name. You'll also learn how to secure your application using Okta's Spring Boot starter and Angular SDK. Below is a screenshot of the app when it's completed.



You will need Java 11 and Node.js 10+ installed to complete this tutorial.

Build an API with Spring Boot 2.2

To get started with Spring Boot 2.2, head on over to start.spring.io and create a new project that uses Java 11 (under more options). Spring



Create a directory to hold your server and client applications. I called mine

```
okta-spring-boot-2-angular-8-example
```

, but you can call yours whatever you like.

> *If you'd rather see the app running than write code, you can see the example on GitHub, or clone and run locally using the commands below.*

```
git clone https://github.com/oktadeveloper/okta-spring-boot-2-angular-8-example.git
cd okta-spring-boot-2-angular-8-example/client
npm install
ng serve &
cd ../server
./mvnw spring-boot:run
```

After downloading

```
demo.zip
```

from start.spring.io, expand it and copy the

```
demo
```

directory to your app-holder directory. Rename

```
demo
```

to

```
server
```

. Open

```
server/pom.xml
```

and comment out the dependency on Okta's Spring Boot starter.

```
<!--dependency>
    <groupId>com.okta.spring</groupId>
    <artifactId>okta-spring-boot-starter</artifactId>
    <version>1.1.0</version>
</dependency-->
```

Open the project in your favorite IDE and create a

```
Car.java
```

class in the

```
src/main/java/com/okta/developer/demo
```

directory. You can use Lombok's annotations to reduce boilerplate code.

```
package com.okta.developer.demo;
```

```
    @Id @GeneratedValue
    private Long id;
    private @NonNull String name;
}
```

Create a `CarRepository` class to perform CRUD (create, read, update, and delete) on the `Car` entity.

```
package com.okta.developer.demo;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource
interface CarRepository extends JpaRepository<Car, Long> {
}
```

Add an `ApplicationRunner` bean to the `DemoApplication` class (in `src/main/java/com/okta/developer/demo/DemoApplication.java`) and use it to add some default data to the database.

```
package com.okta.developer.demo;

import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import java.util.stream.Stream;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

    @Bean
    ApplicationRunner init(CarRepository repository) {
        return args -> {
            Stream.of("Ferrari", "Jaguar", "Porsche", "Lamborghini", "Bugatti",
                      "AMC Gremlin", "Triumph Stag", "Ford Pinto", "Yugo GV").forEach(name -> {
                Car car = new Car();
                car.setName(name);
                repository.save(car);
            });
            repository.findAll().forEach(System.out::println);
        };
    }
}
```

If you start your app (using `./mvnw spring-boot:run`) after adding this code, you'll see the list of cars displayed in your console on startup.

```
Car(id=1, name=Ferrari)
Car(id=2, name=Jaguar)
```

```
Fatal error compiling: invalid target release: 11
```

, it's because you're using Java 8. If you change to use Java 11, this error will go away. If you're using SDKMAN, run

```
sdk install java 11.0.2-open
```

followed by

```
sdk default java 11.0.2-open
```

.

Add a

```
CoolCarController
```

class (in

```
src/main/java/com/okta/developer/demo
```

) that returns a list of cool cars to display in the Angular client.

```
package com.okta.developer.demo;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import java.util.Collection;
import java.util.stream.Collectors;

@RestController
class CoolCarController {
    private CarRepository repository;

    public CoolCarController(CarRepository repository) {
        this.repository = repository;
    }

    @GetMapping("/cool-cars")
    public Collection<Car> coolCars() {
        return repository.findAll().stream()
                .filter(this::isCool)
                .collect(Collectors.toList());
    }

    private boolean isCool(Car car) {
        return !car.getName().equals("AMC Gremlin") &&
                !car.getName().equals("Triumph Stag") &&
                !car.getName().equals("Ford Pinto") &&
                !car.getName().equals("Yugo GV");
    }
}
```

If you restart your server and hit

```
http://localhost:8080/cool-cars
```

with your browser, or a command-line client, you should see the filtered list of cars.

```
$ http :8080/cool-cars
HTTP/1.1 200
Content-Type: application/json;charset=UTF-8
Date: Tue, 07 May 2019 18:07:33 GMT
Transfer-Encoding: chunked

[
    {
        "id": 1,
        "name": "Ferrari"
    },
    {
        "id": 2,
        "name": "Jaguar"
    },
    {
        "id": 3,
        "name": "Porsche"
```

```
]
```

Create a Client with Angular CLI

Angular CLI is a command-line utility that can generate an Angular project for you. Not only can it create new projects, but it can also generate code. It's a convenient tool because it also offers commands that will build and optimize your project for production. It uses webpack under the covers for building.

Install the latest version of Angular CLI (which is version v8.0.0-rc.3 at the time of this writing).

```
npm i -g @angular/cli@v8.0.0-rc.3
```

Create a new project in the umbrella directory you created.

```
ng new client --routing --style css --enable-ivy
```

After the client is created, navigate into its directory, remove

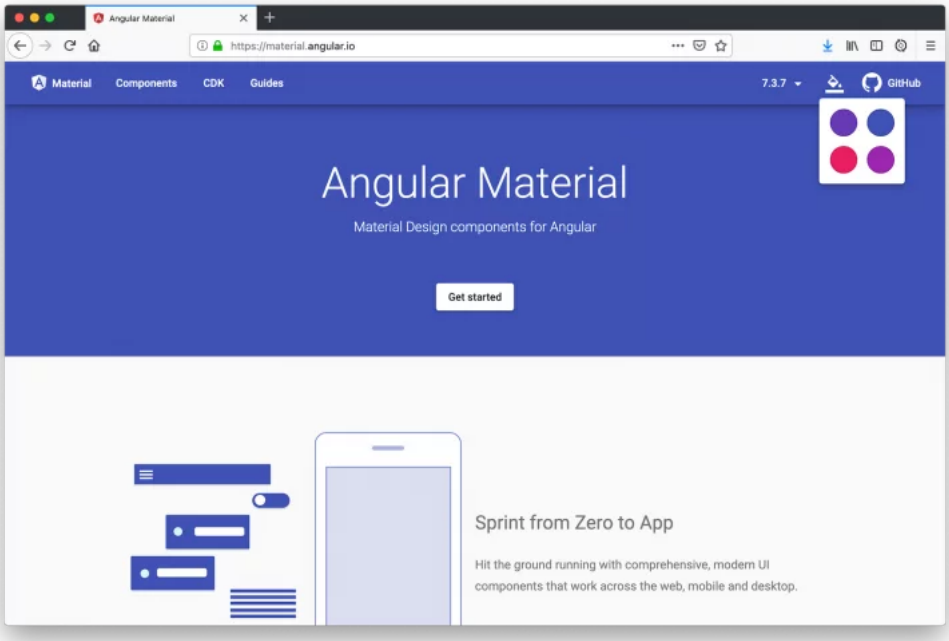
```
.git
```

, and install Angular Material.

```
cd client
rm -rf .git # optional: .git won't be created if you don't have Git installed
ng add @angular/material
```

When prompted for the theme and other options, select the defaults.

You'll use Angular Material's components to make the UI look better, especially on mobile phones. If you'd like to learn more about Angular Material, see material.angular.io. It has extensive documentation on its various components and how to use them. The paint bucket in the top right corner will allow you to preview available theme colors.



Build a Car List Page with Angular CLI

Use Angular CLI to generate a car service that can talk to the Cool Cars API.

```
ng g s shared/car/car
```

Update the code in

```
client/src/app/shared/car/car.service.ts
```

to fetch the list of cars from the server.

```
  constructor(private http: HttpClient) {
  }

  getAll(): Observable<any> {
    return this.http.get('//localhost:8080/cool-cars');
  }
}
```

Open

```
src/app/app.module.ts
```

, and add

```
HttpClientModule
```

as an import.

```
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserAnimationsModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

Generate a

```
car-list
```

component to display the list of cars.

```
ng g c car-list
```

Update

```
client/src/app/car-list/car-list.component.ts
```

to use the

```
CarService
```

to fetch the list and set the values in a local

```
cars
```

variable.

```
import { Component, OnInit } from '@angular/core';
import { CarService } from '../shared/car/car.service';

@Component({
  selector: 'app-car-list',
  templateUrl: './car-list.component.html',
  styleUrls: ['./car-list.component.css']
})
export class CarListComponent implements OnInit {
  cars: Array<any>;

  constructor(private carService: CarService) { }

  ngOnInit() {
    this.carService.getAll().subscribe(data => {
      this.cars = data;
    });
  }
}
```

```
  {{car.name}}
</div>
```

Update

`client/src/app/app.component.html`

to have the

`app-car-list`

element.

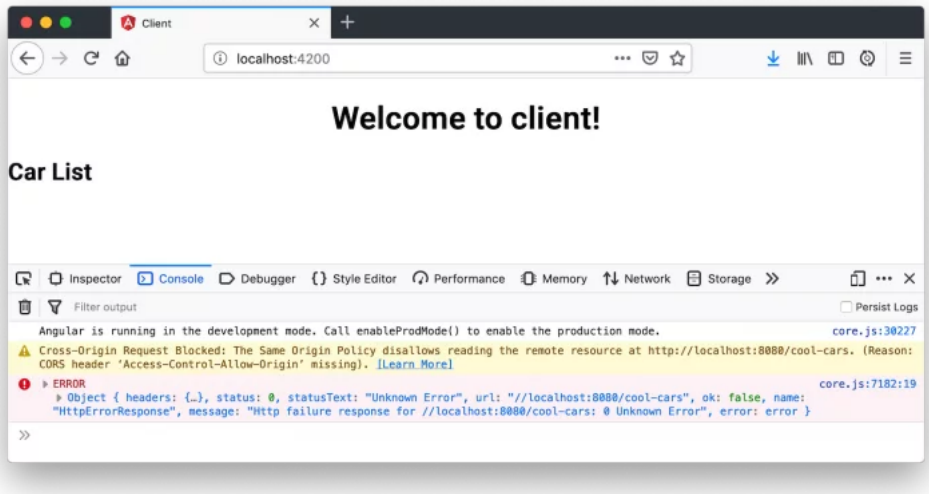
```
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
</div>

<app-car-list></app-car-list>
<router-outlet></router-outlet>
```

Start the client application using

`ng serve -o`

. You won't see the car list just yet, and if you open your developer console, you'll see why.



This error happens because you haven't enabled CORS (Cross-Origin Resource Sharing) on the server.

Enable CORS on the Server

To enable CORS on the server, add a

`@CrossOrigin`

annotation to the

`CoolCarController`

(in

`server/src/main/java/com/okta/developer/demo/CoolCarController.java`

).

```
import org.springframework.web.bind.annotation.CrossOrigin;
...
@GetMapping("/cool-cars")
@CrossOrigin(origins = "http://localhost:4200")
public Collection<Car> coolCars() {
    return repository.findAll().stream()
```

`CarRepository`

. This would allow you to communicate with its endpoints when adding/deleting/editing from Angular.

```
import org.springframework.web.bind.annotation.CrossOrigin;

@RepositoryRestResource
@CrossOrigin(origins = "http://localhost:4200")
interface CarRepository extends JpaRepository<Car, Long> {
}
```

However, this no longer works in Spring Boot 2.2.0.M2. The good news is there is a workaround. You can add a `CorsFilter`

bean to your `DemoApplication.java`

class. This is necessary when you integrate Spring Security as well; you're just doing it a bit earlier.

```
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.core.Ordered;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;
import org.springframework.web.filter.CorsFilter;
import java.util.Collections;

...

public class DemoApplication {
    // main() and init() methods

    @Bean
    public FilterRegistrationBean<CorsFilter> simpleCorsFilter() {
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        CorsConfiguration config = new CorsConfiguration();
        config.setAllowCredentials(true);
        config.setAllowedOrigins(Collections.singletonList("http://localhost:4200"));
        config.setAllowedMethods(Collections.singletonList("*"));
        config.setAllowedHeaders(Collections.singletonList("*"));
        source.registerCorsConfiguration("/**", config);
        FilterRegistrationBean<CorsFilter> bean = new FilterRegistrationBean<>(new CorsFilter(source));
        bean.setOrder(Ordered.HIGHEST_PRECEDENCE);
        return bean;
    }
}
```

Restart the server, refresh the client, and you should see the list of cars in your browser.

Add Angular Material

You've already installed Angular Material, to use its components, you need to import them. Open `client/src/app/app.module.ts`

and add imports for animations, and Material's toolbar, buttons, inputs, lists, and card layout.

```
import { MatButtonModule, MatCardModule, MatInputModule, MatListModule, MatToolbarModule } from '@angular/material';

@NgModule({
  ...
  imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserAnimationsModule,
    HttpClientModule,
    MatButtonModule,
    MatCardModule,
    MatInputModule,
    MatListModule,
    MatToolbarModule
  ],
```

```
<mat-toolbar color="primary">
  <span>Welcome to {{title}}!</span>
</mat-toolbar>

<app-car-list></app-car-list>
<router-outlet></router-outlet>
```

Update

`client/src/app/car-list/car-list.component.html`

to use the card layout and list component.

```
<mat-card>
  <mat-card-title>Car List</mat-card-title>
  <mat-card-content>
    <mat-list>
      <mat-list-item *ngFor="let car of cars">
        <img mat-list-avatar src="{{car.giphyUrl}}" alt="{{car.name}}">
        <h3 mat-line>{{car.name}}</h3>
      </mat-list-item>
    </mat-list>
  </mat-card-content>
</mat-card>
```

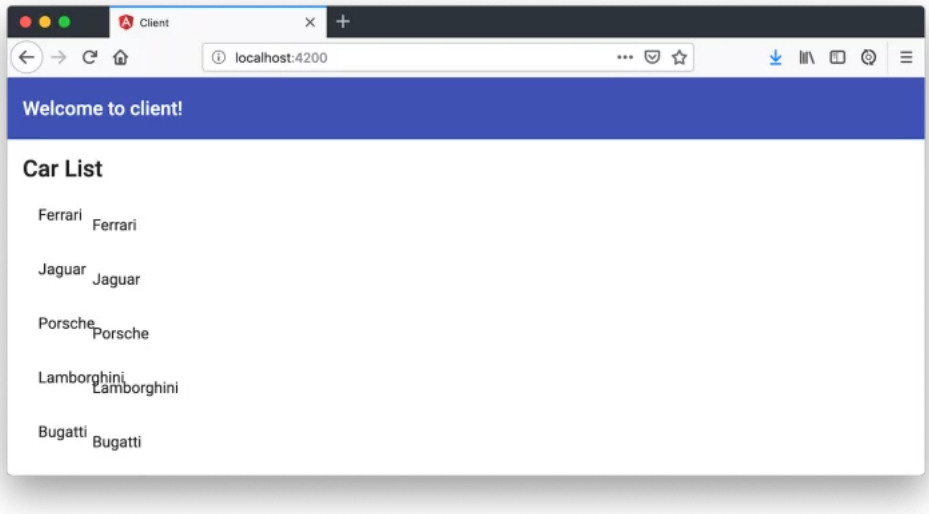
If you run your client with

`ng serve`

and navigate to

`http://localhost:4200`

, you'll see the list of cars, but no images associated with them.



Add Animated GIFs with Giphy

To add a

`giphyUrl`

property to each car, create

`client/src/app/shared/giphy/giphy.service.ts`

and populate it with the code below.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { map } from 'rxjs/operators';

@Injectable({providedIn: 'root'})
```

```
    const apiLink = this.giphyApi + searchTerm;
    return this.http.get(apiLink).pipe(map((response: any) => {
      if (response.data.length > 0) {
        return response.data[0].images.original.url;
      } else {
        return 'https://media.giphy.com/media/YaOxRsmrv9IeA/giphy.gif'; // dancing cat for 404
      }
    }));
  }
}
```

Update the code in

```
client/src/app/car-list/car-list.component.ts
```

to set the

```
giphyUrl
```

property on each car.

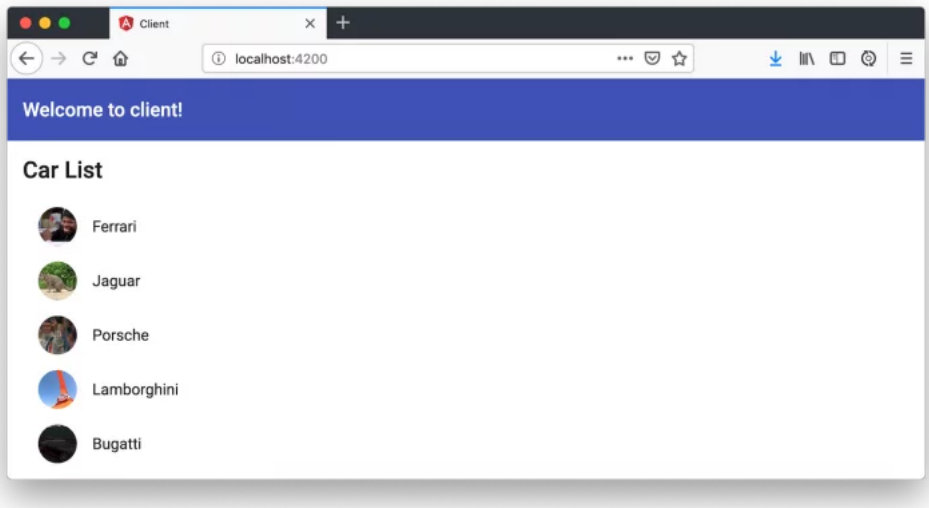
```
import { GiphyService } from '../shared/giphy/giphy.service';

export class CarListComponent implements OnInit {
  cars: Array<any>;

  constructor(private carService: CarService, private giphyService: GiphyService) { }

  ngOnInit() {
    this.carService.getAll().subscribe(data => {
      this.cars = data;
      for (const car of this.cars) {
        this.giphyService.get(car.name).subscribe(url => car.giphyUrl = url);
      }
    });
  }
}
```

Now your browser should show you the list of car names, along with an avatar image beside them.



Add an Edit Feature to Your Angular App

Having a list of car names and images is cool, but it's a lot more fun when you can interact with it! To add an edit feature, start by generating a

```
car-edit
```

component.

```
ng g c car-edit
```

```
@RepositoryRestResource
```

annotation.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({providedIn: 'root'})
export class CarService {
  public API = '//localhost:8080';
  public CAR_API = this.API + '/cars';

  constructor(private http: HttpClient) {
  }

  getAll(): Observable<any> {
    return this.http.get(this.API + '/cool-cars');
  }

  get(id: string) {
    return this.http.get(this.CAR_API + '/' + id);
  }

  save(car: any): Observable<any> {
    let result: Observable<any>;
    if (car.href) {
      result = this.http.put(car.href, car);
    } else {
      result = this.http.post(this.CAR_API, car);
    }
    return result;
  }

  remove(href: string) {
    return this.http.delete(href);
  }
}
```

In

```
client/src/app/car-list/car-list.component.html
```

, add a link to the edit component. Also, add a button at the bottom to add a new car.

```
<mat-card>
  <mat-card-title>Car List</mat-card-title>
  <mat-card-content>
    <mat-list>
      <mat-list-item *ngFor="let car of cars">
        <img mat-list-avatar src="{{car.giphyUrl}}" alt="{{car.name}}">
        <h3 mat-line>
          <a mat-button [routerLink]="['/car-edit', car.id]">{{car.name}}</a>
        </h3>
      </mat-list-item>
    </mat-list>
  </mat-card-content>

  <button mat-fab color="primary" [routerLink]="['/car-add']">Add</button>
</mat-card>
```

In

```
client/src/app/app.module.ts
```

, import the

```
FormsModule
```

.

```
import { FormsModule } from '@angular/forms';

@NgModule({
```

`client/src/app/app-routing.module.ts`

, add routes for the

`CarListComponent`

and

`CarEditComponent`

.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { CarListComponent } from './car-list/car-list.component';
import { CarEditComponent } from './car-edit/car-edit.component';

const routes: Routes = [
  { path: '', redirectTo: '/car-list', pathMatch: 'full' },
  {
    path: 'car-list',
    component: CarListComponent
  },
  {
    path: 'car-add',
    component: CarEditComponent
  },
  {
    path: 'car-edit/:id',
    component: CarEditComponent
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Modify

`client/src/app/car-edit/car-edit.component.ts`

to fetch a car's information from the id passed on the URL, and to add methods for saving and deleting.

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { Subscription } from 'rxjs';
import { ActivatedRoute, Router } from '@angular/router';
import { CarService } from '../shared/car/car.service';
import { GiphyService } from '../shared/giphy/giphy.service';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'app-car-edit',
  templateUrl: './car-edit.component.html',
  styleUrls: ['./car-edit.component.css']
})
export class CarEditComponent implements OnInit, OnDestroy {
  car: any = {};

  sub: Subscription;

  constructor(private route: ActivatedRoute,
              private router: Router,
              private carService: CarService,
              private giphyService: GiphyService) {
  }

  ngOnInit() {
    this.sub = this.route.params.subscribe(params => {
      const id = params.id;
      if (id) {
        this.carService.get(id).subscribe((car: any) => {
          if (car) {
```

```
    });
  }

  ngOnDestroy() {
    this.sub.unsubscribe();
  }

  gotoList() {
    this.router.navigate(['/car-list']);
  }

  save(form: NgForm) {
    this.carService.save(form).subscribe(result => {
      this.gotoList();
    }, error => console.error(error));
  }

  remove(href) {
    this.carService.remove(href).subscribe(result => {
      this.gotoList();
    }, error => console.error(error));
  }
}
```

Update the HTML in

`client/src/app/car-edit/car-edit.component.html`

to have a form with the car's name, as well as to display the image from Giphy.

```
<mat-card>
  <form #carForm="ngForm" (ngSubmit)="save(carForm.value)">
    <mat-card-header>
      <mat-card-title><h2>{{car.name ? 'Edit' : 'Add'}} Car</h2></mat-card-title>
    </mat-card-header>
    <mat-card-content>
      <input type="hidden" name="href" [(ngModel)]="car.href">
      <mat-form-field>
        <input matInput placeholder="Car Name" [(ngModel)]="car.name"
               required name="name" #name>
      </mat-form-field>
    </mat-card-content>
    <mat-card-actions>
      <button mat-raised-button color="primary" type="submit"
              [disabled]="!carForm.valid">Save</button>
      <button mat-raised-button color="secondary" (click)="remove(car.href)"
              *ngIf="car.href" type="button">Delete</button>
      <a mat-button routerLink="/car-list">Cancel</a>
    </mat-card-actions>
    <mat-card-footer>
      <div class="giphy">
        <img src="{{car.giphyUrl}}" alt="{{car.name}}">
      </div>
    </mat-card-footer>
  </form>
</mat-card>
```

Put a little padding around the image by adding the following CSS to

`client/src/app/car-edit/car-edit.component.css`

.

```
.giphy {
  margin: 10px;
}
```

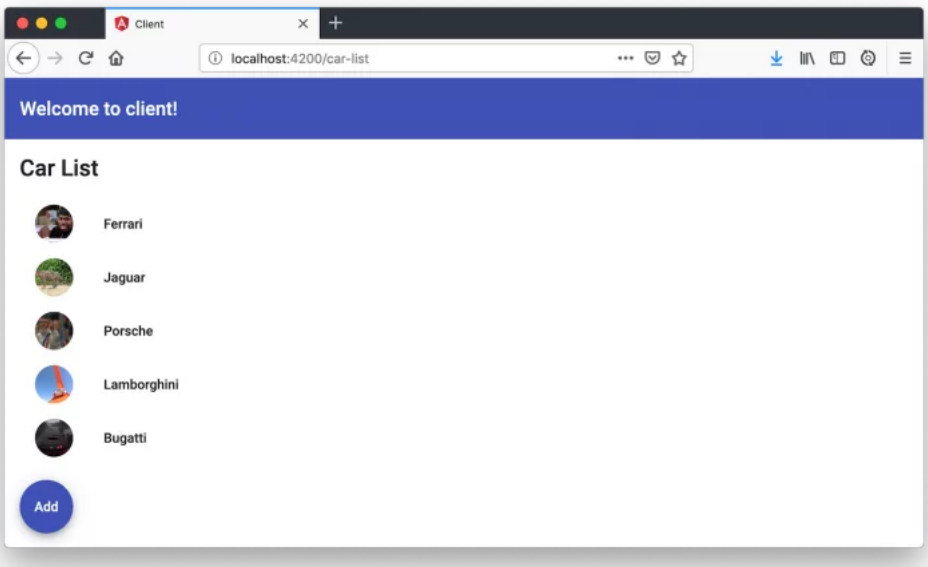
Modify

`client/src/app/app.component.html`

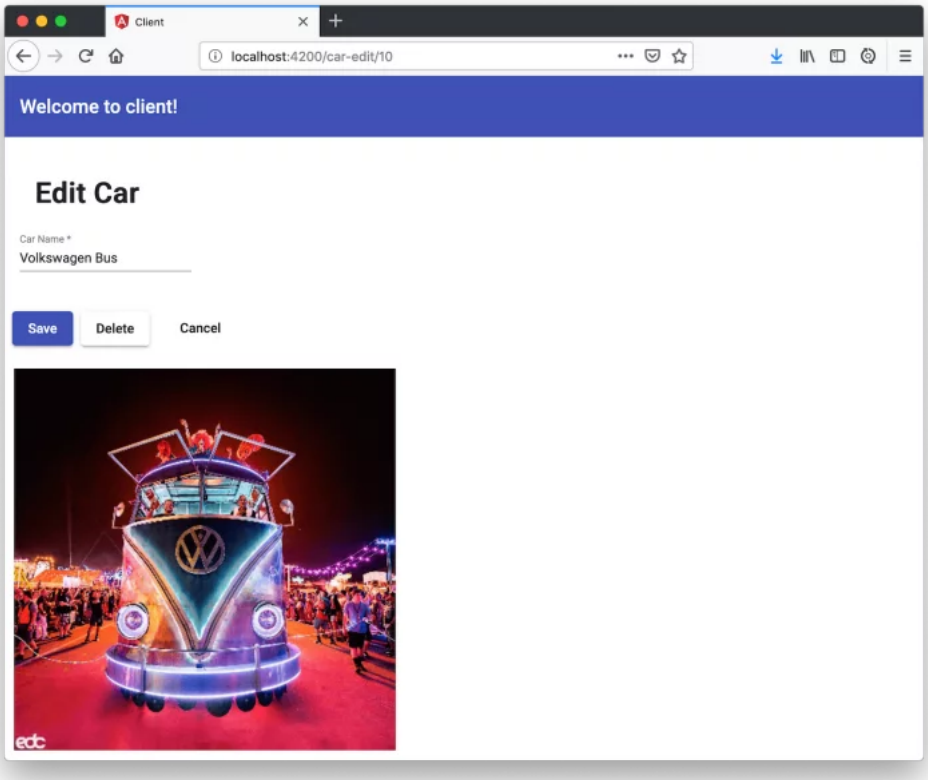
and remove

```
<app-car-list></app-car-list>
```

button.



The following screenshot shows what it looks like to edit a car that you've added.



Add OIDC Authentication to Your Spring Boot + Angular App

Add authentication with OIDC is a nifty feature you can add to this application. Knowing who the person is can come in handy if you want to add auditing, or personalize your application (with a rating feature for example).

Spring Security + OIDC

```
    <version>1.1.0</version>
</dependency>
```

Now you need to configure the server to use Okta for authentication. You'll need to create an OIDC app in Okta for that.

Create an OIDC App in Okta

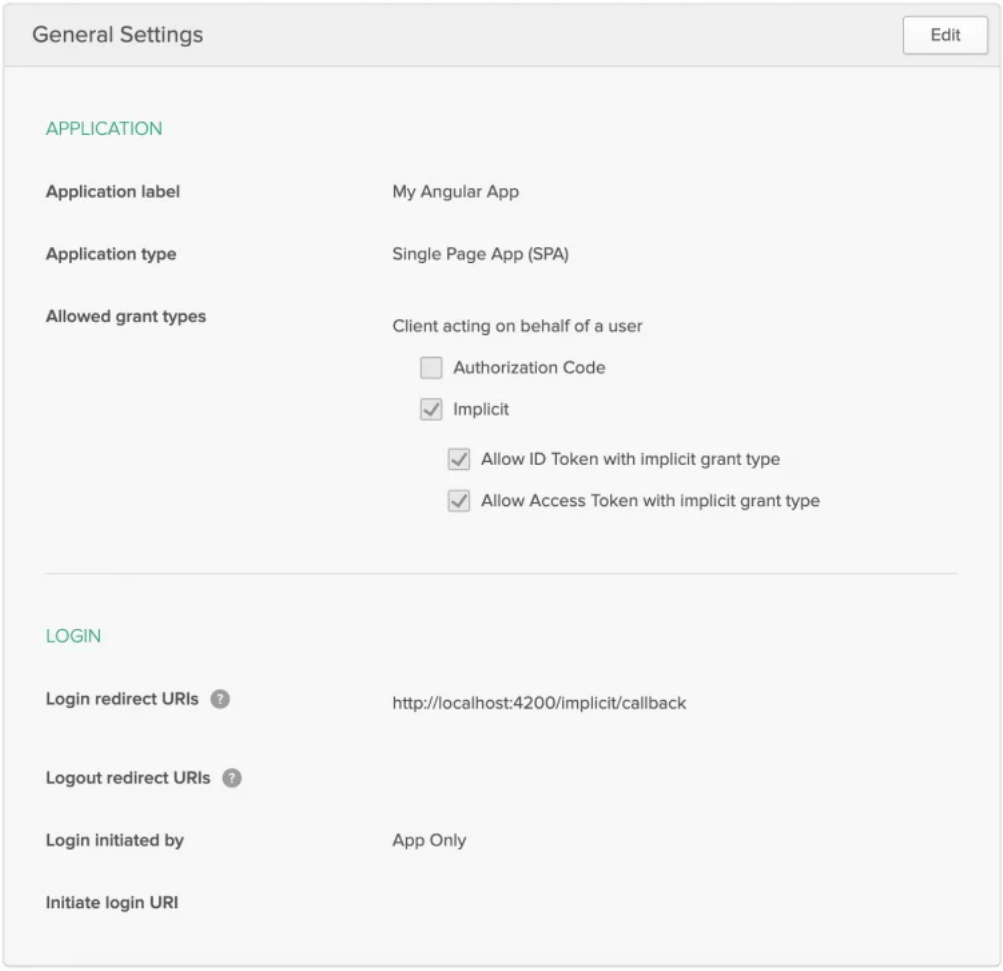
Log in to your Okta Developer account (or sign up if you don't have an account) and navigate to **Applications** > **Add Application**. Click **Single-Page App**, click **Next**, and give the app a name you'll remember. Change all instances of

```
http://localhost:8080
```

to

```
http://localhost:4200
```

and click **Done**.



You'll see a client ID at the bottom of the page. Add it and an

```
issuer
```

property to

```
server/src/main/resources/application.properties
```

.

```
okta.oauth2.client-id={yourClientId}
okta.oauth2.issuer=https://{yourOktaDomain}/oauth2/default
```

Create

```
server/src/main/java/com/okta/developer/demo/SecurityConfiguration.java
```

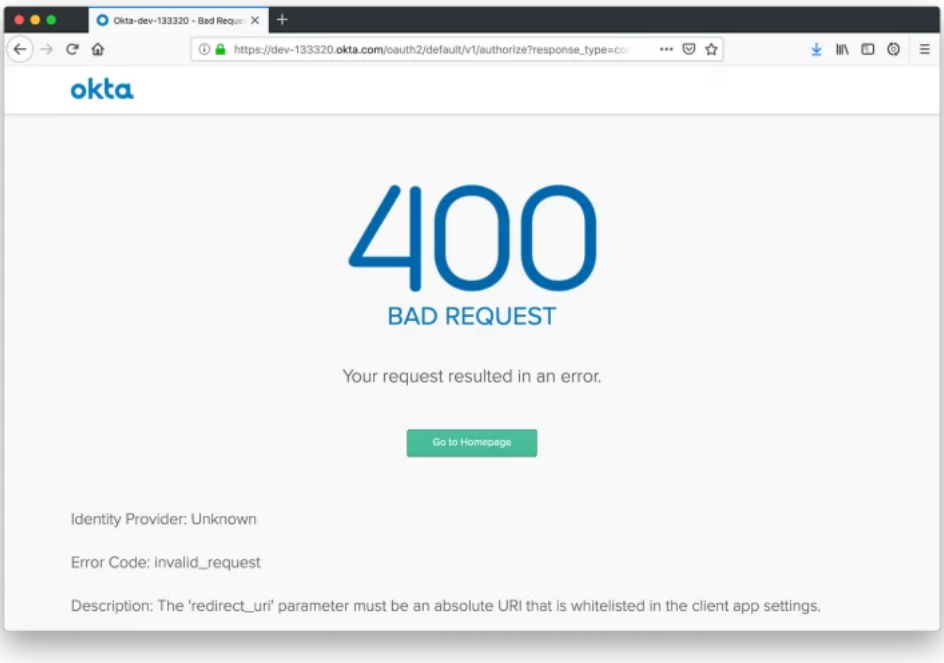
```
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests().anyRequest().authenticated()
            .and()
            .oauth2ResourceServer().jwt();
    }
}
```

After making these changes, you should be able to restart your app and see an error when you try to navigate to

```
http://localhost:8080
```

.



NOTE: You could fix this error by adding

```
http://localhost:8080/login/oauth2/code/okta
```

as a redirect URI to your app, but it won't solve the problem. If you want to support OIDC Login with Spring Boot, you'll need to register a **Web** app (instead of a SPA) and include a client secret in your

```
application.properties
```

. This is not a necessary step in this tutorial.

Now that your server is locked down, you need to configure your client to talk to it with an access token. This is where Okta's Angular SDK comes in handy.

Okta's Angular Support

The Okta Angular SDK is a wrapper around Okta Auth JS, which builds on top of OIDC. More information about Okta's Angular library can be found on GitHub.

To simplify our Angular SDK's installation and configuration, we created an @oktadev/schematics project that does everything for you. You can read more about how @oktadev/schematics works in Use Angular Schematics to Simplify Your Life.

```
git commit -m "Initialize project"
```

To install and configure Okta's Angular SDK, run the following command in the `client` directory:

```
ng add @oktadev/schematics --issuer=https://{yourOktaDomain}/oauth2/default --clientId={yourClientId}
```

This command will:

- Install `@okta/okta-angular`
- Configure Okta's Angular SDK for your app in `auth-routing.module.ts`
- Add `isAuthenticated` logic to `app.component.ts`
- Add a `HomeComponent` with login and logout buttons
- Configure routing with a default route to `/home` and an `/implicit/callback` route
- Add an `HttpInterceptor` that adds an `Authorization` header with an access token to `localhost` requests

Modify `client/src/app/app.component.html` to use Material components and to have a logout button.

```
<mat-toolbar color="primary">
  <span>Welcome to {{title}}!</span>
  <span class="toolbar-spacer"></span>
  <button mat-raised-button color="accent" *ngIf="isAuthenticated"
          (click)="oktaAuth.logout()" [routerLink]="['/home']">Logout
  </button>
</mat-toolbar>

<router-outlet></router-outlet>
```

You might notice there's a span with a `toolbar-spacer` class. To make that work as expected, add a `toolbar-spacer`

Then update

`client/src/app/home/home.component.html`

to use Angular Material and link to the Car List.

```
<mat-card>
  <mat-card-content>
    <button mat-raised-button color="accent" *ngIf="!isAuthenticated"
            (click)="oktaAuth.loginRedirect()">Login
    </button>
    <button mat-raised-button color="accent" *ngIf="isAuthenticated"
            [routerLink]="['/car-list']">Car List
    </button>
  </mat-card-content>
</mat-card>
```

Since you're using Material components in

`HomeComponent`

, which is managed by the newly-added

`client/src/app/auth-routing.module.ts`

, you'll need to import

`MatCardModule`

.

```
import { MatCardModule } from '@angular/material';

@NgModule({
  ...
  imports: [
    ...
    MatCardModule
  ],
  ...
})
```

To make it so there's not a bottom border at the bottom of your content, make the

`<mat-card>`

element fill the screen by adding the following to

`client/src/styles.css`

.

```
mat-card {
  height: 100vh;
}
```

Now if you restart your client, everything *should* work. Unfortunately, it does not because Ivy does not yet implement CommonJS/UMD support. As a workaround, you can modify

`tsconfig.app.json`

to disable Ivy.

```
"angularCompilerOptions": {
  "enableIvy": false
}
```

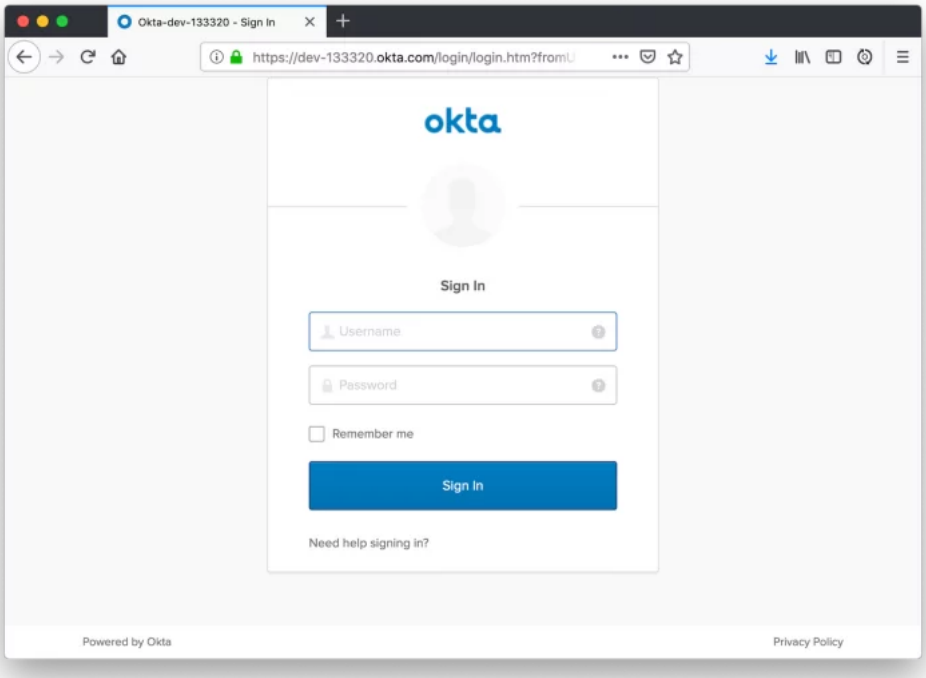
Stop and restart the

`ng serve`

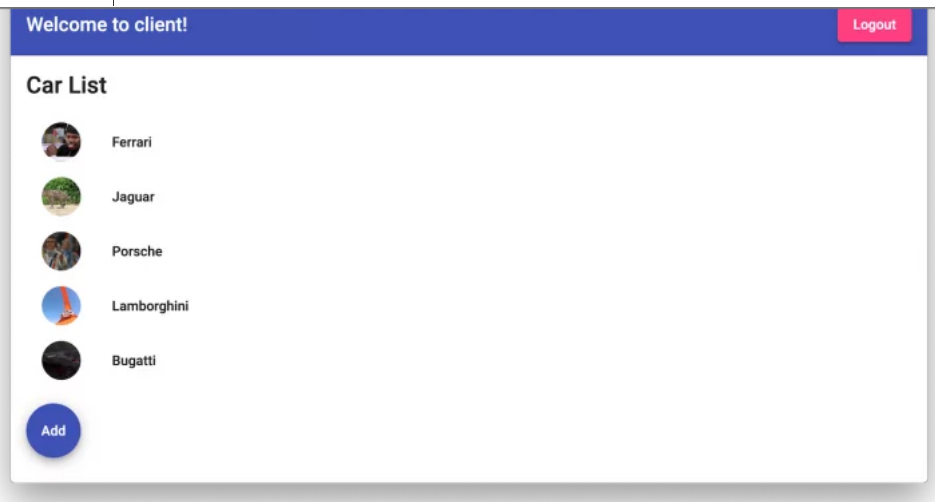
process. Open your browser to

`http://localhost:4200`

Click on the **Login** button. If you've configured everything correctly, you'll be redirected to Okta to log in.



Enter valid credentials, and you should be redirected back to your app. Celebrate when it all works! 🚀



Learn More about Spring Boot and Angular

It can be tough to keep up with fast-moving frameworks like Spring Boot and Angular. This post is meant to give you a jump start on the latest releases. For specific changes in Angular 8, see the Angular team's plan for version 8.0 and Ivy. For Spring Boot, see its 2.2 Release Notes.

You can see the full source code for the application developed in this tutorial on GitHub at oktadeveloper/okta-spring-boot-2-angular-8-example.

This blog has a plethora of Spring Boot and Angular tutorials. Here are some of my favorites:

- Build a Desktop Application with Angular and Electron
- Migrate Your Spring Boot App to the Latest and Greatest Spring Security and OAuth 2.0
- i18n in Java 11, Spring Boot, and JavaScript
- Build Secure Login for Your Angular App
- Build Reactive APIs with Spring WebFlux

If you have any questions, please don't hesitate to leave a comment below, or ask us on our Okta Developer Forums. Don't forget to follow us on Twitter and YouTube too!

"Angular 8 + Spring Boot 2.2: Build a CRUD App Today!" was originally published on the Okta developer blog on May 13, 2019.

Tagged with: ANGULAR SPRING SPRING BOOT

LIKE THIS ARTICLE? READ MORE FROM JAVA CODE GEEKS

2 Leave a Reply

Join the discussion...

1 1 0

2

This site uses Akismet to reduce spam. Learn how your comment data is processed.

Subscribe

newest oldest most voted

Author

Are you talking about @RepositoryRestResource? If I remove it, you won't be able to PUT, POST, or DELETE to /cars. I believe it's necessary.

+ 0 − Reply

6 days ago

KNOWLEDGE BASE

Courses
Examples
Minibooks
Resources
Tutorials

PARTNERS

Mkyong

THE CODE GEEKS NETWORK

.NET Code Geeks
Java Code Geeks
System Code Geeks
Web Code Geeks

HALL OF FAME

"Android Full Application Tutorial" series
11 Online Learning websites that you should check out
Advantages and Disadvantages of Cloud Computing – Cloud computing pros and cons
Android Google Maps Tutorial
Android JSON Parsing with Gson Tutorial
Android Location Based Services Application – GPS location
Android Quick Preferences Tutorial
Difference between Comparator and Comparable in Java
GWT 2 Spring 3 JPA 2 Hibernate 3.5 Tutorial
Java Best Practices – Vector vs ArrayList vs HashSet

ABOUT JAVA CODE GEEKS

JCGs (Java Code Geeks) is an independent online community focused o
ultimate Java to Java developers resource center; targeted at the techn
technical team lead (senior developer), project manager and junior dev
JCGs serve the Java, SOA, Agile and Telecom communities with daily n
domain experts, articles, tutorials, reviews, announcements, code snipp
source projects.

DISCLAIMER