How to Use Spring Retry

by Chris Shayan · Jul. 12, 18 · Java Zone · Tutorial

A few days ago, I noticed that there is a group of people asking how to use Spring Retry. Before I go into the sample code, let me quickly explain the purpose behind Spring Retry. Spring Retry provides the ability to automatically re-invoke a failed operation. This is helpful when errors may be transient in nature (like a momentary network glitch). Spring Retry provides a declarative control of the process and policy-based behavior that is easy to extend and customize.

You can find the complete source code in here.

Maven Dependencies

```
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.retry</groupId>
<artifactId>spring-retry</artifactId>
</dependency>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-aop</artifactId>
</dependency>
<dependency>
<groupId>org.aspectj</groupId>
<artifactId>aspectjweaver</artifactId>
</dependency>

<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

Enable Retry

```
package com.chrisshayan.example.springretry;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.retry.annotation.EnableRetry;

@EnableRetry
@SpringBootApplication
public class SpringRetryApplication {

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

}
```

Using Retry With Annotations

```
package com.chrisshayan.example.springretry;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.retry.annotation.Backoff;
import org.springframework.retry.annotation.Recover;
import org.springframework.retry.annotation.Retryable;
import org.springframework.stereotype.Service;

@Service
public class SampleRetryService {
    private static final Logger LOGGER = LoggerFactory.getLogger(SampleRetryService.class);

    private static int COUNTER = 0;

    @Retryable(
            value = {TypeOneException.class, TypeTwoException.class},
            maxAttempts = 4, backoff = @Backoff(2000))
    public String retryWhenException() throws TypeOneException, TypeTwoException {
        COUNTER++;
        LOGGER.info("COUNTER = " + COUNTER);

        if(COUNTER == 1)
            throw new TypeOneException();
        else if(COUNTER == 2)
            throw new TypeTwoException();
        else
            throw new RuntimeException();
    }

    @Recover
```

```
    @Recover
    public String recover(Throwable t) {
        LOGGER.info("SampleRetryService.recover");
        return "Error Class :: " + t.getClass().getName();
    }
}
```

In order for your test class to work, the retry needs to be in the proper context. This is because we need to have another service that wraps around the retry. This is called:

```
package com.chrisshayan.example.springretry;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class SampleRetryClientService {

    @Autowired
    private SampleRetryService sampleRetryService;


    public String callRetryService() throws TypeOneException, TypeTwoException {
        return sampleRetryService.retryWhenException();
    }
}
```

Test Class

```
package com.chrisshayan.example.springretry;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest
public class SpringRetryApplicationTests {

    private static final Logger LOGGER = LoggerFactory.getLogger(SpringRetryApplicationTests.class);
    @Autowired
    private SampleRetryClientService client;

@Test
public void contextLoads() {
}

@Test
    public void sampleRetryService() {
```

```
        try {
            final String message = client.callRetryService();
            LOGGER.info("message = " + message);
        } catch (TypeOneException | TypeTwoException e) {
            e.printStackTrace();
        }
    }

}
```

Console

```
2018-07-10 23:42:45.528  INFO 14583 --- [           main] c.c.e.springretry.SampleRetryService     : C
2018-07-10 23:42:47.534  INFO 14583 --- [           main] c.c.e.springretry.SampleRetryService     : C
2018-07-10 23:42:49.538  INFO 14583 --- [           main] c.c.e.springretry.SampleRetryService     : C
2018-07-10 23:42:49.539  INFO 14583 --- [           main] c.c.e.springretry.SampleRetryService     : S
2018-07-10 23:42:49.539  INFO 14583 --- [           main] c.c.e.s.SpringRetryApplicationTests      : m
```

There are more capabilities in Spring Retry, such as Stateless Retry, Stateful Retry, and different retry policies and listeners. You can read more here.

Like This Article? Read More From DZone

Using Spring Data JPA Specification

Spring XML-Based DI and Builder Pattern

Command Patterns in Spring Framework

Free DZone Refcard
Java 13

Topics: SPRING , RETRY , RETRY PATTERN , JAVA , TUTORIAL

Redis Cluster on Java for Scaling and High Availability

by Nikita Koksharov · Sep 20, 19 · Java Zone · Tutorial

What Is a Redis Cluster?

Scalability and availability are two of the most important qualities of any enterprise-class database.

It's very unusual that you can exactly predict the maximum amount of resources your database will consume, so scalability is a must in order to deal with periods of unexpectedly high demand. Yet, scalability is useless without availability, which ensures that users will always be able to access the information within the database when they need it.

You may also like: Creating Distributed Java Applications With Redis

---

Redis is an in-memory data structure store that can be used to implement a non-relational key-value database. However, the bare-bones installation of Redis does not come equipped for maximum performance right off the bat.

In order to improve the scalability and availability of a Redis deployment, you can use Redis Cluster, a method of automatically sharding data across different Redis nodes. Redis Cluster breaks up a very large Redis database into smaller horizontal partitions known as shards that are stored on separate servers.

This makes the Redis database able to accommodate a larger number of requests, and therefore more scalable. What's more, availability improves because the database can continue operations even when some of the nodes in the cluster have failed.

Before version 3.0, Redis Cluster used asynchronous replication. This meant that, in practice, if a master in Redis Cluster crashes before sending a write to all of its slaves, one of the slaves that did not receive the write can be promoted to master, which will cause the write to be lost.

Since version 3.0, Redis Cluster also has a synchronous replication option in the form of the `WAIT` command, which blocks the current client until all write commands are successfully completed. While this is not enough to guarantee strong consistency, it does make the data transfer process significantly more secure.

How to Run a Redis Cluster

There are two ways to get Redis Cluster up and running: the easy way and the hard way.

The easy way involves using the `create-cluster` bash script, which you can find in the `utils/create-cluster` directory in your Redis installation. The following two commands will create a default cluster with 6 nodes, 3 masters, and 3 slaves:

```
create-cluster start
create-cluster create
```

Once the cluster is created, you can interact with it. By default, the first node in the cluster starts at port 30001. Stop the cluster with the command:

```
create-cluster stop
```

The hard way of running Redis Cluster involves setting up your own configuration files for the cluster. All instances of Redis Cluster must contain at least three master nodes.

Below is an example configuration file for a single node:

```
port 7000
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
appendonly yes
```

As its name suggests, the `cluster-enabled` option enables the cluster mode. The `cluster-config-file` option contains a path to the configuration file for the given node.

To create a test Redis Cluster instance with three master nodes and three slave nodes, execute the following commands in your terminal:

```
mkdir cluster-test
cd cluster-test
mkdir 7000 7001 7002 7003 7004 7005
```

Within each of these six directories, create a **redis.conf** configuration file using the example configuration file given above. Then, copy your **redis-server** executable into the **cluster-test** directory and use it to launch six different nodes in six different tabs of your terminal.

Connecting to Redis Cluster on Java

Like the base Redis installation, Redis Cluster is not able to work with the Java programming language out of the box. The good news is that there are frameworks that make it easy for you to use Redis Cluster and Java together.

Redisson is a Java client for Redis that includes many common constructs in Java, including a variety of objects, collections, locks, and services. Because Redisson reimplements these constructs in a distributed fashion, they can be

collections, locks, and services. Because Redisson reimplements these constructs in a distributed fashion, they can be shared across multiple applications and servers, allowing them to work with tools like Redis Cluster.

The following code demonstrates the usage of Redisson with Redis Cluster:

```
package redis.demo;
import org.redisson.Redisson;
import org.redisson.api.RBucket;
import org.redisson.api.RedissonClient;

/**
 * Redis Sentinel Java example
 *
 */
public class Application
{
    public static void main( String[] args )
    {
        Config config = new Config();
        config.useClusterServers()
              .addNodeAddress("redis://127.0.0.1:6379", "redis://127.0.0.1:6380");

        RedissonClient redisson = Redisson.create(config);

        // operations with Redis based Lock

        // implements java.util.concurrent.locks.Lock
        RLock lock = redisson.getLock("simpleLock");
        lock.lock();

        try {
           // do some actions
        } finally {
           lock.unlock();
        }

        // operations with Redis based Map

        // implements java.util.concurrent.ConcurrentMap
        RMap<String, String> map = redisson.getMap("simpleMap");
        map.put("mapKey", "This is a map value");

        String mapValue = map.get("mapKey");
        System.out.println("stored map value: " + mapValue);

        redisson.shutdown();
    }
}
```

Redisson is an open-source client that enables Java programmers to work with Redis with minimal stress and complication, making the development process drastically easier and more familiar.

Further Reading

Further Reading

Creating Distributed Java Applications With Redis

Quickstart: How to Use Redis on Java

Java Distributed Caching in Redis

Like This Article? Read More From DZone

Redisson PRO vs. Jedis: Which Is Faster?

How to Connect to Redis on Java Over SSL

Database Caching With Redis and Java

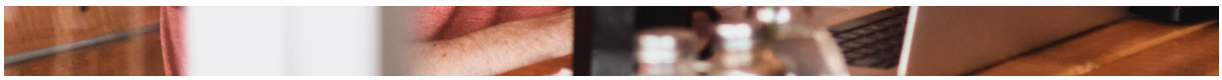
Free DZone Refcard
Java 13

Topics: JAVA, REDIS, CLUSTER, SCALING, TUTORIAL, REDISSON, REDIS CLUSTER

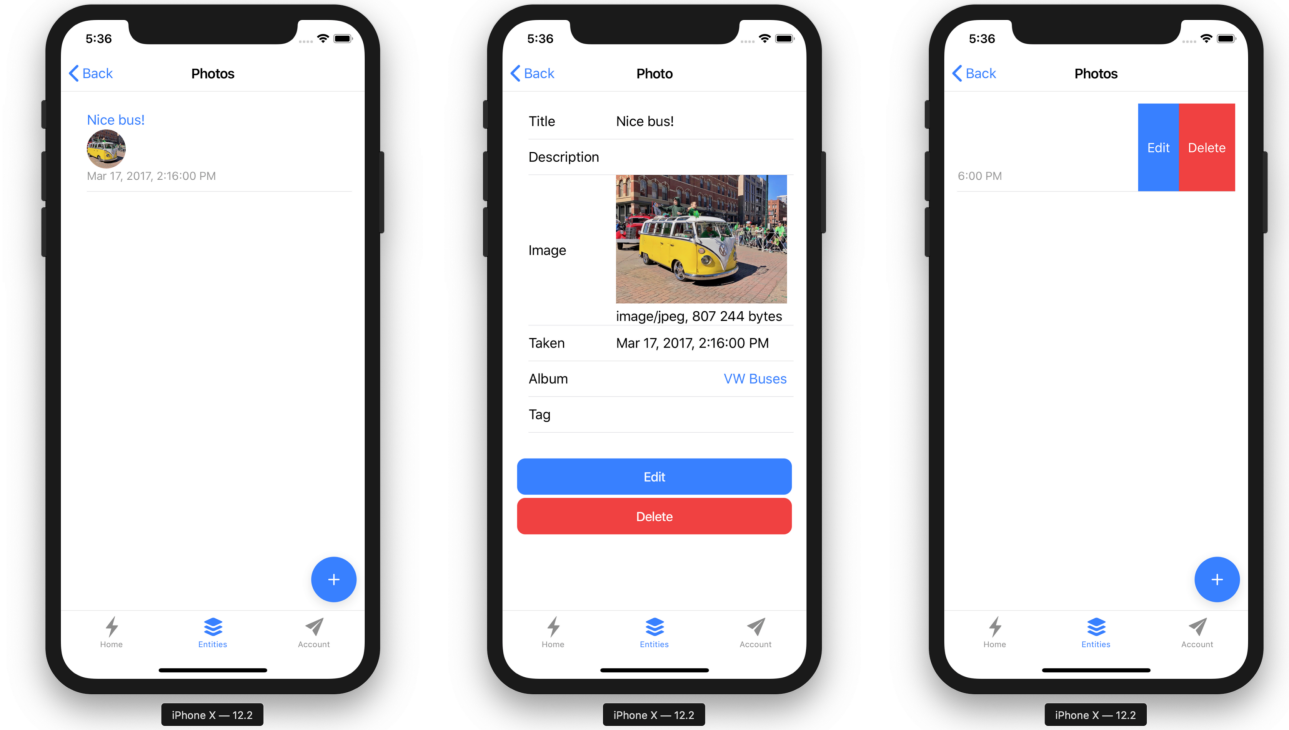
Opinions expressed by DZone contributors are their own.

How to Use Ionic 4 for JHipster 6 to Build a Mobile App

by Matt Raible MVB · **Sep 20, 19 · Java Zone · Tutorial**



Developers were using JHipster for designing mobile apps before 'hipster' was even, like, a thing



For all those who know me, you know how much I love Java, Spring Boot, JHipster, and Ionic.

JHipster is the best thing ever. It's a popular, fully open-source app generator and platform where you can quickly build Java apps with JavaScript front-ends.

Ionic is also a complete open-source framework where you can build cross-platform apps (hybrid) with web technology. Hybrid mobile apps, similar to native mobile apps, can be listed on app stores and installed on your mobile devices. Ionic supports PWA, which means you can ship the same app for the web to an app store.

You may also like: Ionic Framework: Getting Started

Spring Boot is the only back-end framework currently supported, with .NET and Node.js implementations currently in development. On the front-end, Angular, React, Vue, React Native, and Ionic are all supported.

In this brief tutorial, I'll show you to use Ionic for JHipster v4 with Spring Boot and JHipster 6.

To complete this tutorial, you'll need to have Java 8+, Node.js 10+, and Docker installed. You'll also need to create an Okta developer account.

Create a Spring Boot + Angular App With JHipster

You can install JHipster via Homebrew (`brew install jhipster`) or with npm.

```
1  npm i -g generator-jhipster@6.1.2
```

Once you have JHipster installed, you have two choices. There's the quick way to generate an app (which I

recommend, and there's the tedious way of picking all your options. I don't care which one you use, but you **must** select **OAuth 2.0 / OIDC** authentication to complete this tutorial successfully.

Here's the easy way:

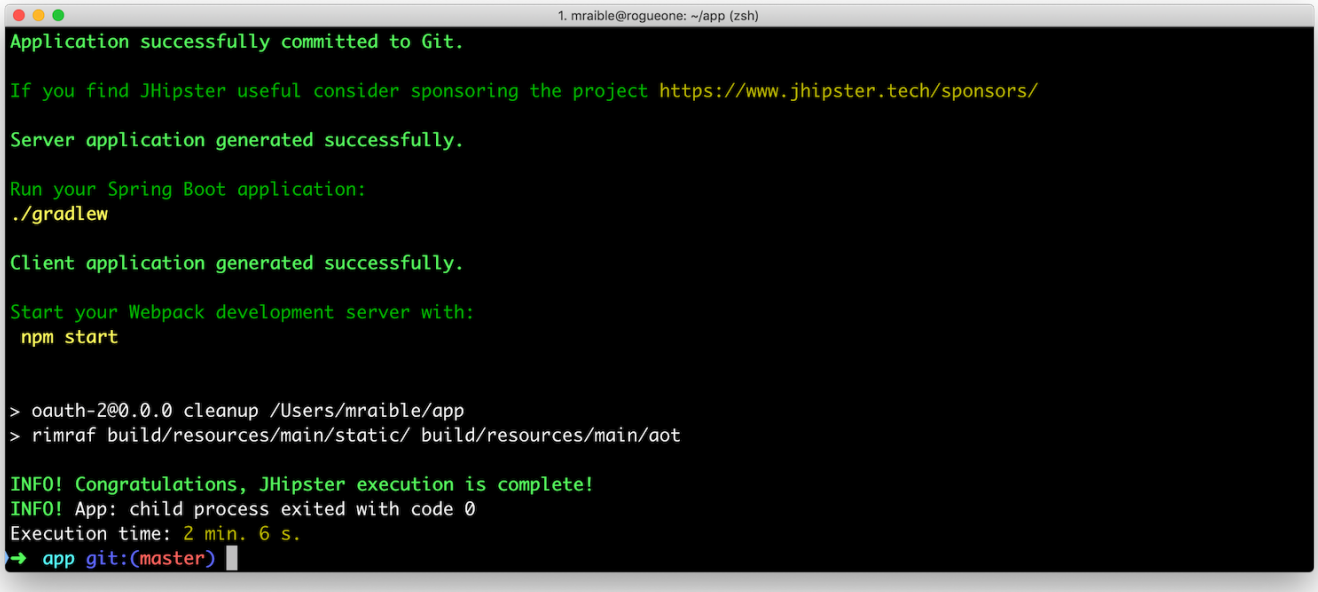
```
mkdir app && cd app

echo "application { config { baseName oauth2, authenticationType oauth2, \
  buildTool gradle, testFrameworks [protractor] }}" >> app.jh

jhipster import-jdl app.jh
```

The hard way is you run `jhipster` and answer a number of questions. There are so many choices when you run this option that you might question your sanity. At last count, I remember reading that JHipster allows 26K+ combinations!

The project generation process will take a couple of minutes to complete if you're on fast Internet and have a bad-ass laptop. When it's finished, you should see output like the following.



OIDC With Keycloak and Spring Security

JHipster has several authentication options: JWT, OAuth 2.0 / OIDC, and UAA. With JWT (the default), you store the access token on the client (in local storage); this works but isn't the most secure. UAA involves using your own OAuth 2.0 authorization server (powered by Spring Security), and OAuth 2.0/OIDC allows you to use Keycloak or Okta.

Spring Security makes Keycloak and Okta integration so incredibly easy it's silly. Keycloak and Okta are called "identity providers," and if you have a similar solution that is OIDC-compliant, I'm confident it'll work with Spring Security and JHipster.

Having Keycloak set by default is nice because you can use it without having an internet connection.

To log into the JHipster app you just created, you'll need to have Keycloak up and running. When you create a JHipster project with OIDC for authentication, it creates a Docker container definition that has the default users and roles. Start Keycloak using the following command.

```
docker-compose -f src/main/docker/keycloak.yml up -d
```

Start your application with `./gradlew` (or `./mvnw` if you chose Maven) and you should be able to log in using "admin/admin" for your credentials.

Open another terminal and prove all the end-to-end tests pass:

```
npm run e2e
```

If your environment is set up correctly, you'll see output like the following:

```
> oauth-2@0.0.0 e2e /Users/mraible/app
> protractor src/test/javascript/protractor.conf.js

[16:02:18] W/configParser - pattern ./e2e/entities/**/*.spec.ts did not match any files.
[16:02:18] I/launcher - Running 1 instances of WebDriver
[16:02:18] I/direct - Using ChromeDriver directly...



  account
    ✓ should fail to login with bad password
    ✓ should login successfully with admin account (1754ms)

  administration
    ✓ should load metrics
    ✓ should load health
    ✓ should load configuration
    ✓ should load audits
    ✓ should load logs



  7 passing (15s)

[16:02:36] I/launcher - 0 instance(s) of WebDriver still running
[16:02:36] I/launcher - chrome #01 passed
Execution time: 19 s.
```

OIDC With Okta and Spring Security

To switch to Okta, you'll first need to create an OIDC app. If you don't have an Okta Developer account, now is the time!

Why Okta instead of Keycloak?

Keycloak works great in development, and Okta has free multi-factor authentication, email support, and excellent performance for production. A developer account gets you 1000 monthly active users for free! You can see other free features and our transparent pricing at developer.okta.com/pricing.

Log in to your Okta Developer account.

- In the top menu, click on **Applications**

- Click on **Add Application**
- Select **Web** and click **Next**
- Enter `JHipster FTW!` for the Name (this value doesn't matter, so feel free to change it)
- Change the Login redirect URI to be `http://localhost:8080/login/oauth2/code/oidc`
- Click **Done**, then **Edit** and add `http://localhost:8080` as a Logout redirect URI
- Click **Save**

These are the steps you'll need to complete for JHipster. Start your JHipster app using a command like the following:

```
SPRING_SECURITY_OAUTH2_CLIENT_PROVIDER_OIDC_ISSUER_URI=https://{yourOktaDomain}/oauth2/default \
  SPRING_SECURITY_OAUTH2_CLIENT_REGISTRATION_OIDC_CLIENT_ID=$clientId \
  SPRING_SECURITY_OAUTH2_CLIENT_REGISTRATION_OIDC_CLIENT_SECRET=$clientSecret ./gradlew
```

The above command can be painful to type, so I encourage you to copy/paste or set the values as environment variables. You can also configure them in a properties/YAML file in Spring Boot, but you should never store secrets in source control.

Create a Native App for Ionic

You'll also need to create a Native app for Ionic. The reason for this is because Ionic for JHipster is configured to use PKCE (Proof Key for Code Exchange). The current Spring Security OIDC support in JHipster still requires a client secret. PKCE does not.

Go back to the Okta developer console and follow the steps below:

- In the top menu, click on **Applications**
- Click on **Add Application**
- Select **Native** and click **Next**
- Enter `Ionic FTW!` for the Name
- Add Login redirect URIs: `http://localhost:8100/implicit/callback` and `dev.localhost.ionic:/callback`
- Click **Done**, then **Edit** and add Logout redirect URIs: `http://localhost:8100/implicit/logout` and `dev.localhost.ionic:/logout`
- Click **Save**

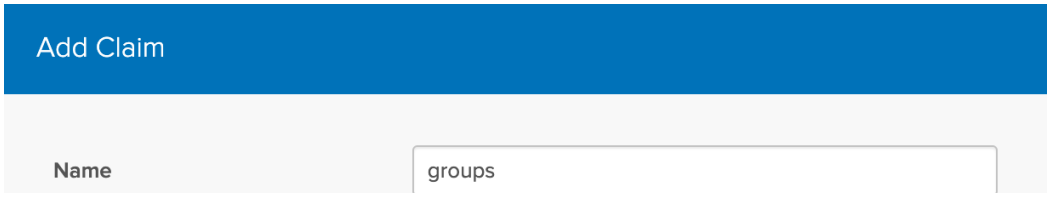
You'll need the client ID from your Native app, so keep your browser tab open or copy/paste it somewhere.

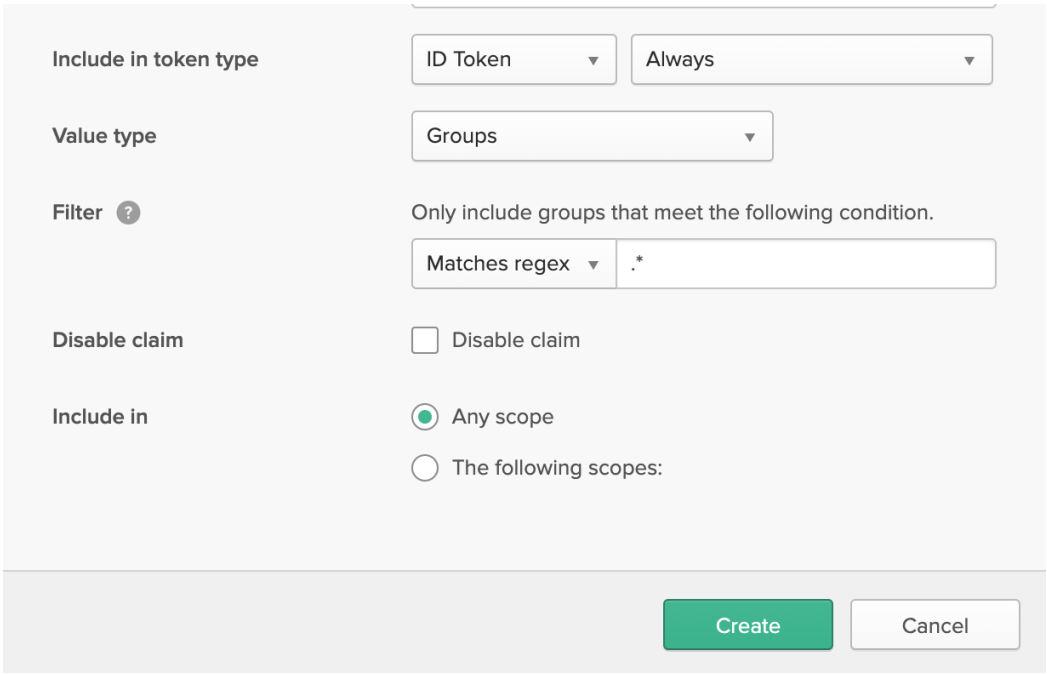
Create Groups and Add Them as Claims to the ID Token

In order to login to your JHipster app, you'll need to adjust your Okta authorization server to include a `groups` claim.

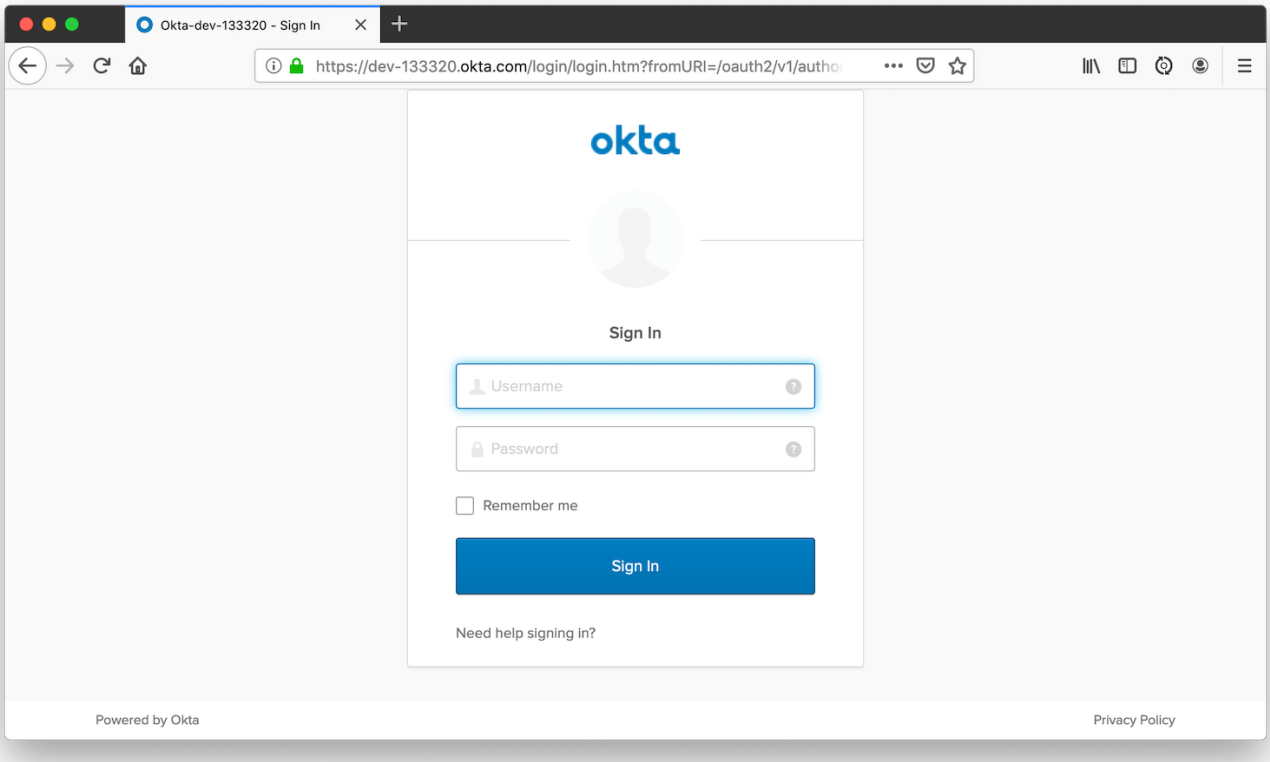
On Okta, navigate to **Users** > **Groups**. Create `ROLE_ADMIN` and `ROLE_USER` groups and add your account to them.

Navigate to **API** > **Authorization Servers**, click the **Authorization Servers** tab and edit the **default** one. Click the **Claims** tab and **Add Claim**. Name it "groups" or "roles" and include it in the ID Token. Set the value type to "Groups" and set the filter to be a Regex of `.*`. Click **Create**.

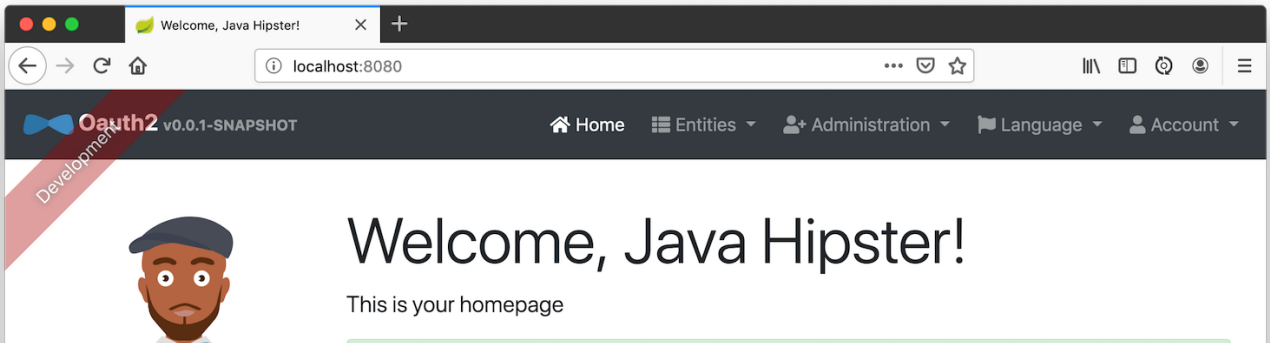


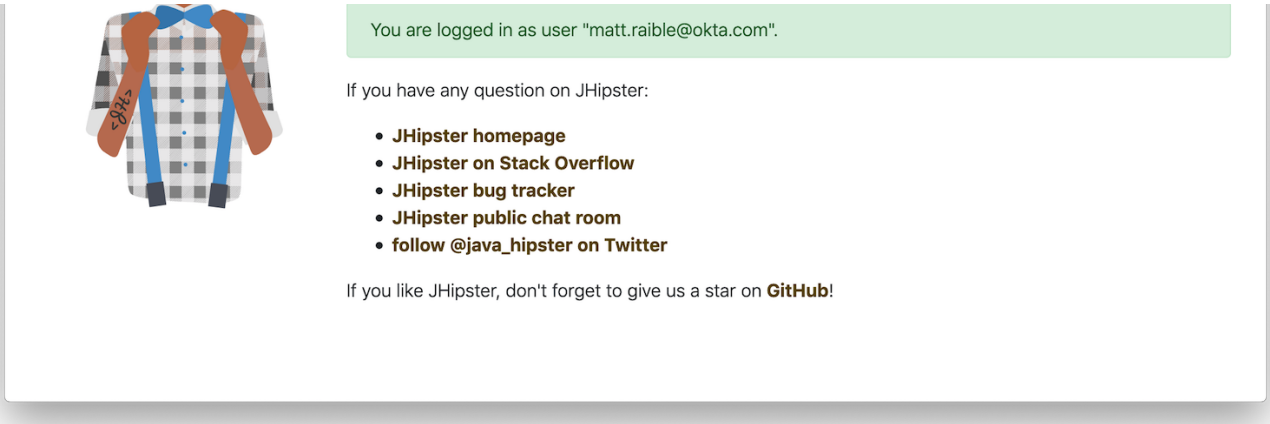


Navigate to `http://localhost:8080` , click **sign in,** and you'll be redirected to Okta to log in.



Enter the credentials you used to sign up for your account, and you should be redirected back to your JHipster app.



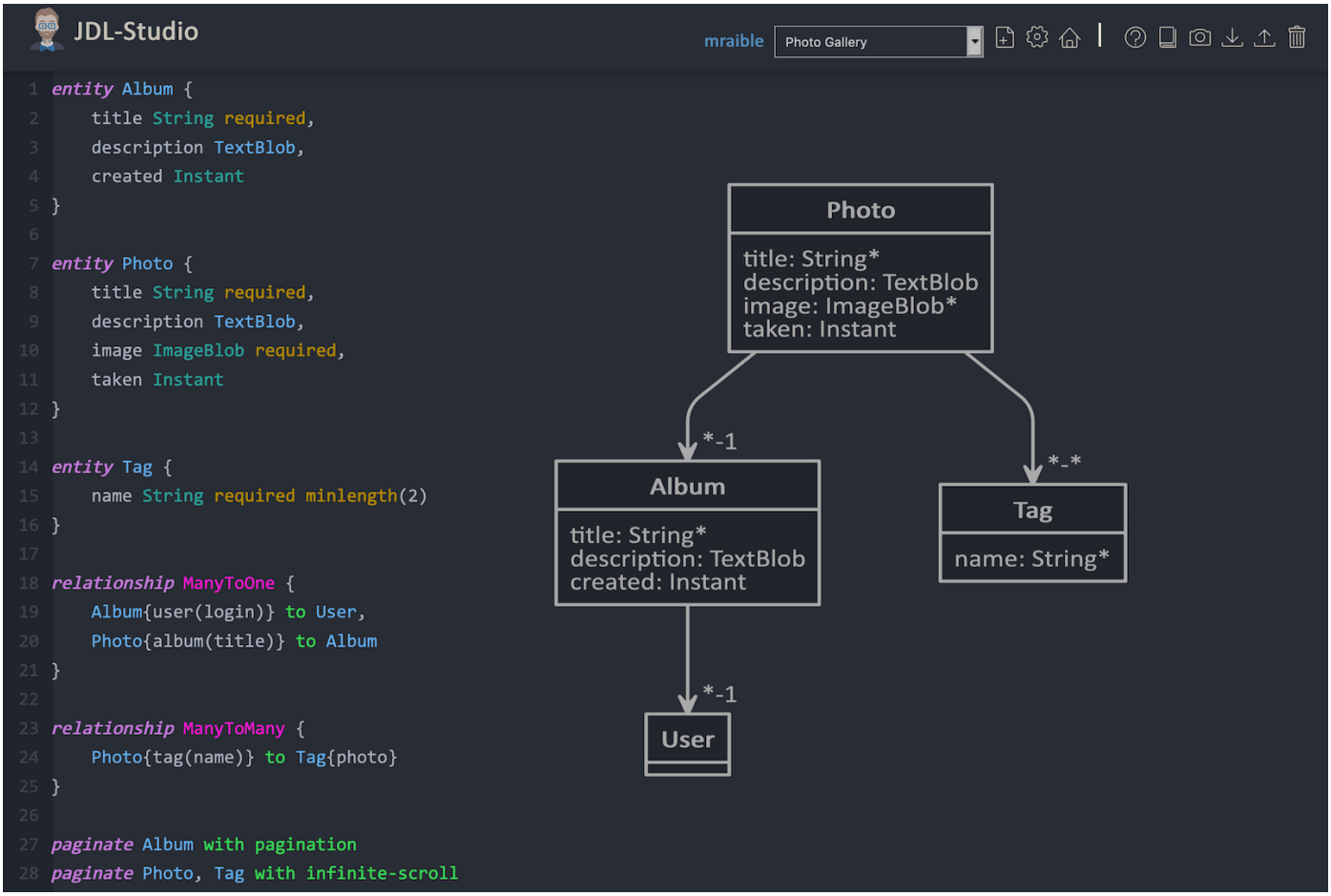


Generate Entities for a Photo Gallery

Let's enhance this example a bit and create a photo gallery that you can upload pictures to. Kinda like Flickr, but waaayyyy more primitive.

JHipster has a JDL (JHipster Domain Language) feature that allows you to model the data in your app, and generate entities from it. You can use its JDL Studio feature to do this online and save it locally once you've finished.

I created a data model for this app that has an `Album`, `Photo`, and `Tag` entities and set up relationships between them. Below is a screenshot of what it looks like in JDL Studio.



CCopy the JDL below and save it in a `photos.jdl` file in the root directory of your project.

```
entity Album {
  title String required,
  description TextBlob,
  created Instant
```

```
}

entity Photo {
  title String required,
  description TextBlob,
  image ImageBlob required,
  taken Instant
}

entity Tag {
  name String required minlength(2)
}

relationship ManyToOne {
  Album{user(login)} to User,
  Photo{album(title)} to Album
}

relationship ManyToMany {
  Photo{tag(name)} to Tag{photo}
}

paginate Album with pagination
paginate Photo, Tag with infinite-scroll
```

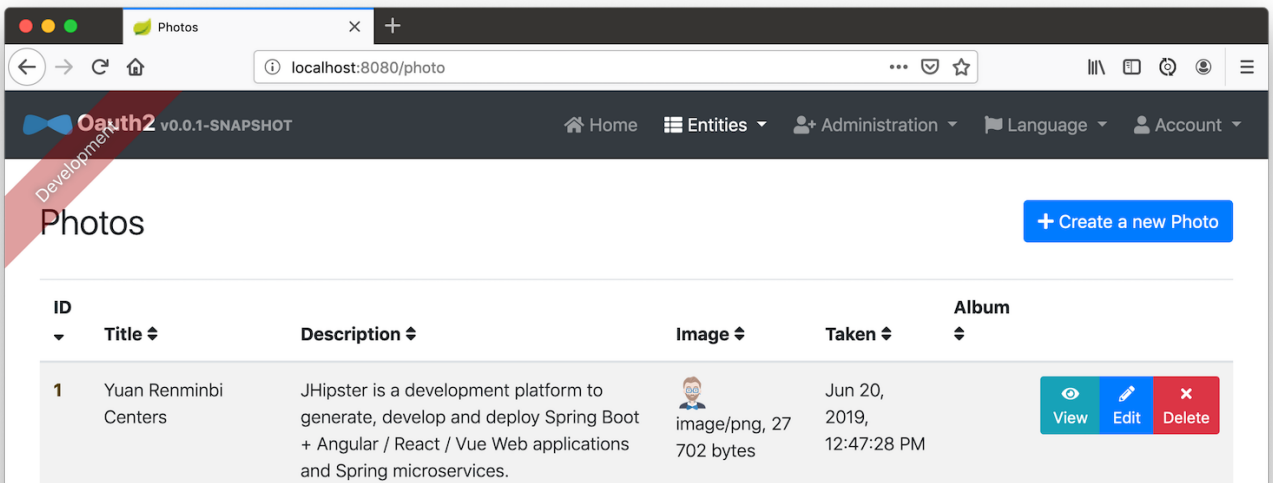
You can generate entities and CRUD code (Java for Spring Boot; TypeScript, and HTML for Angular) using the following command:

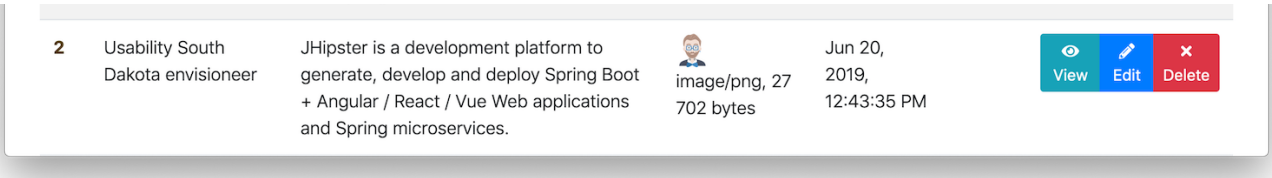
```
jhipster import-jdl photos.jdl
```

When prompted, type **a** to update existing files.

This process will create Liquibase changelog files (to create your database tables), entities, repositories, Spring MVC controllers, and all the Angular code that's necessary to create, read, update, and delete your data objects. It'll even generate Jest unit tests and Protractor end-to-end tests!

When the process completes, restart your app, and confirm that all your entities exist (and work) under the **Entities** menu.





You might notice that the entity list screen is pre-loaded with data. This is done by faker.js. To turn it off, edit `src/main/resources/config/application-dev.yml`, search for `liquibase`, and set its `contexts` value to `dev`. I made this change in this example's code and ran `./gradlew clean` to clear the database.

```
liquibase:
  # Add 'faker' if you want the sample data to be loaded automatically
  contexts: dev
```

Develop a Mobile App With Ionic and Angular

Getting started with Ionic for JHipster is similar to JHipster. You simply have to install the Ionic CLI, Yeoman, the module itself, and run a command to create the app.

```
npm i -g generator-jhipster-ionic@4.0.0 ionic@5.1.0 yo
yo jhipster-ionic
```

If you have your `app` application at `~/app`, you should run this command from your home directory (`~`). Ionic for JHipster will prompt you for the location of your backend application. Use `mobile` for your app's name and `app` for the JHipster app's location.

Type **a** when prompted to overwrite `mobile/src/app/app.component.ts`.

Open `mobile/src/app/auth/auth.service.ts` in an editor, search for `data.clientId`, and replace it with the client ID from your Native app on Okta.

```
// try to get the oauth settings from the server
this.requestor.xhr({method: 'GET', url: AUTH_CONFIG_URI}).then(async (data: any) => {
  this.authConfig = {
    identity_client: '{yourClientId}',
    identity_server: data.issuer,
    redirect_url: redirectUri,
    end_session_redirect_url: logoutRedirectUri,
    scopes,
    usePkce: true
  };
  ...
}
```

When using Keycloak, this change is not necessary.

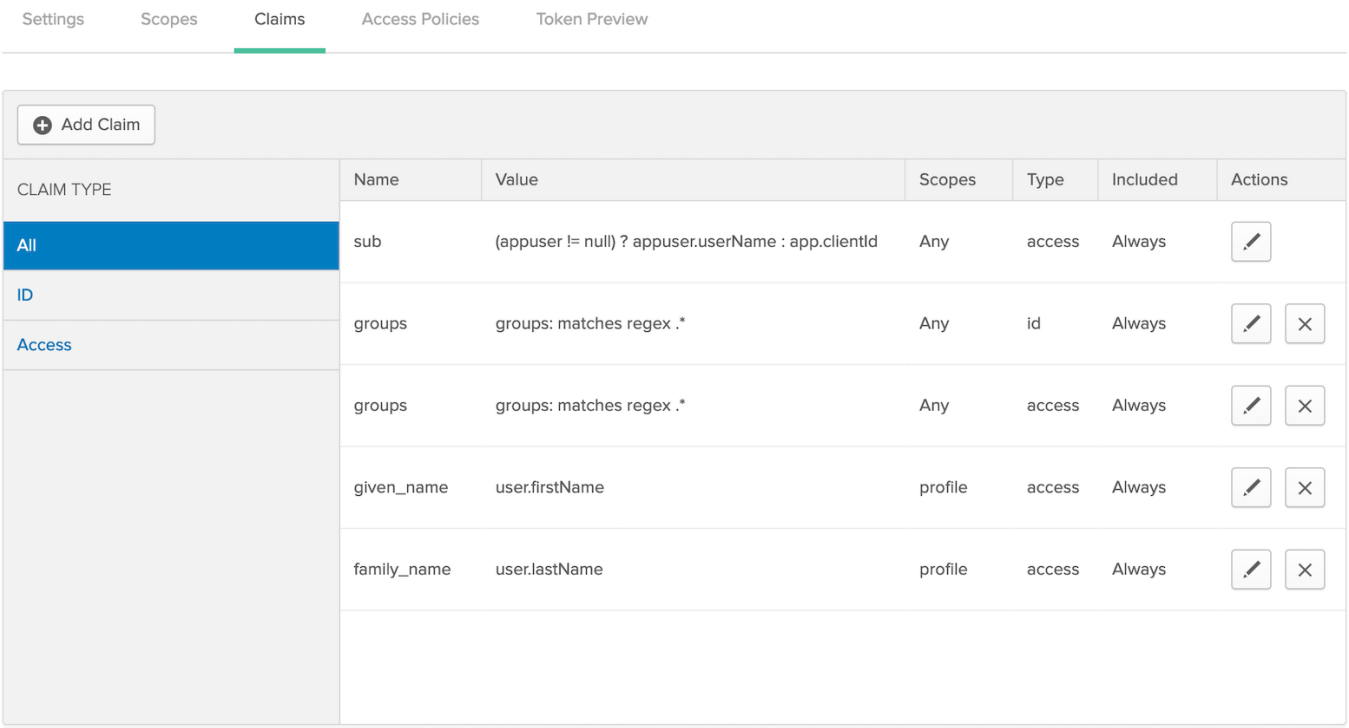
Add Claims to Access Token

In order to set-up authentication successfully with your Ionic app, you have to do a bit more configuration in Okta. Since the Ionic client will only send an access token to JHipster, you need to 1) add a `groups` claim to the access token and 2) add a couple more claims so the user's name will be available in JHipster.

Navigate to **API** > **Authorization Servers**, click the **Authorization Servers** tab, and edit the **default** one. Click the **Claims** tab and **Add Claim**. Name it "groups" and include it in the Access Token. Set the value type to "Groups" and set the filter to be a Regex of `.*`. Click **Create**.

Add another claim, name it `given_name`, include it in the access token, use `Expression` in the value type, and set the value to `user.firstName`. Optionally, include it in the `profile` scope. Perform the same actions to create a `family_name` claim and use the expression `user.lastName`.

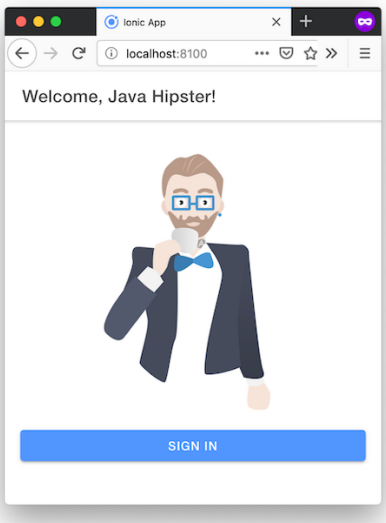
When you are finished, your claims should look as follows.

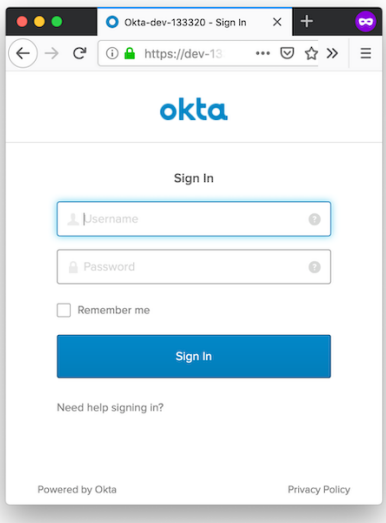


Run the following commands to start your Ionic app.

```
cd mobile
ionic serve
```

You'll see a screen with a sign-in button. Click on it, and you'll be redirected to Okta to authenticate.





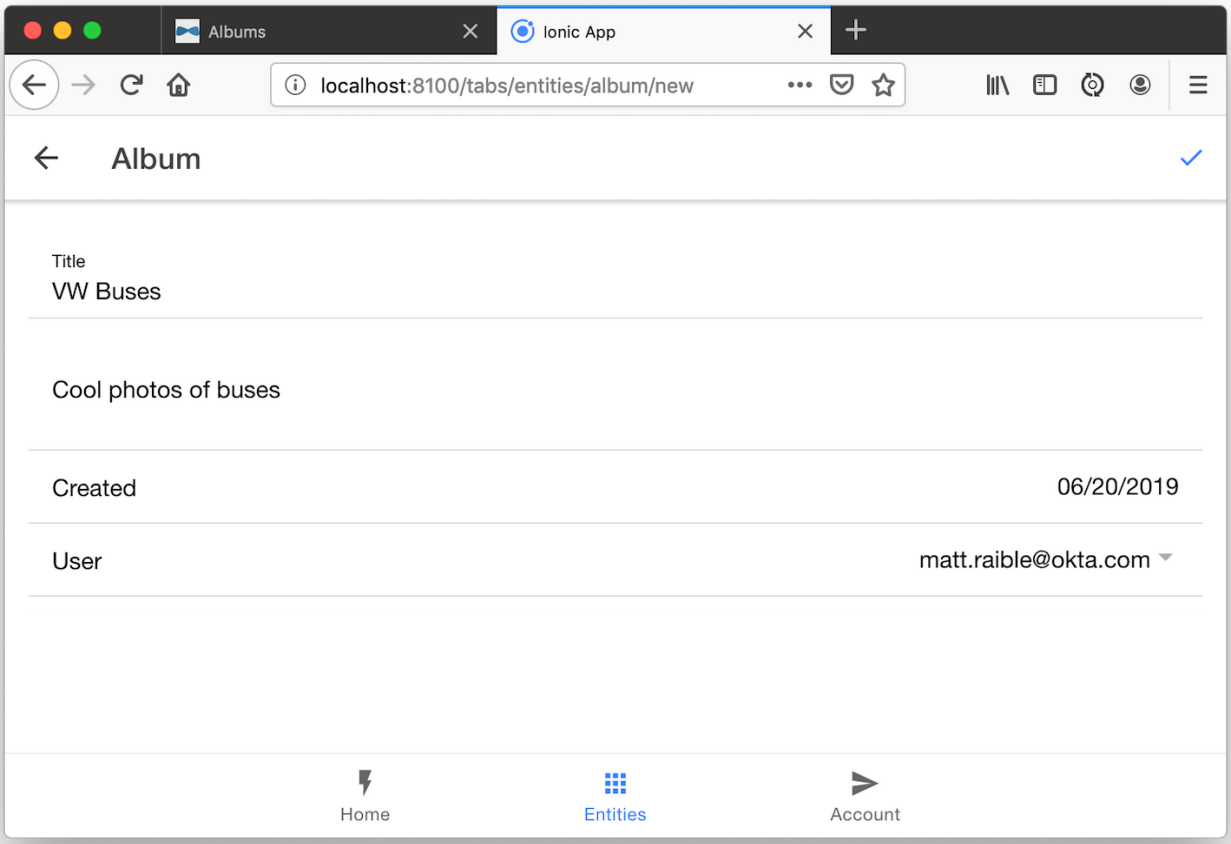
Now that you having log in working, you can use the entity generator to generate Ionic pages for your data model. Run

the following commands (in your `~/mobile` directory) to generate screens for your entities.

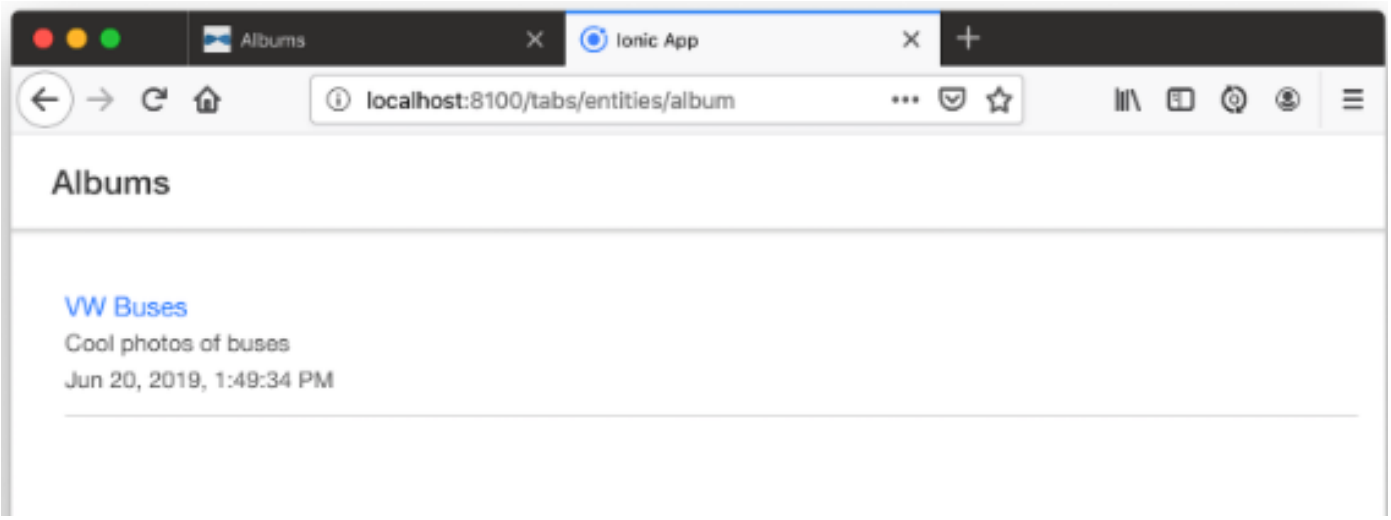
```
yo jhipster-ionic:entity album
```

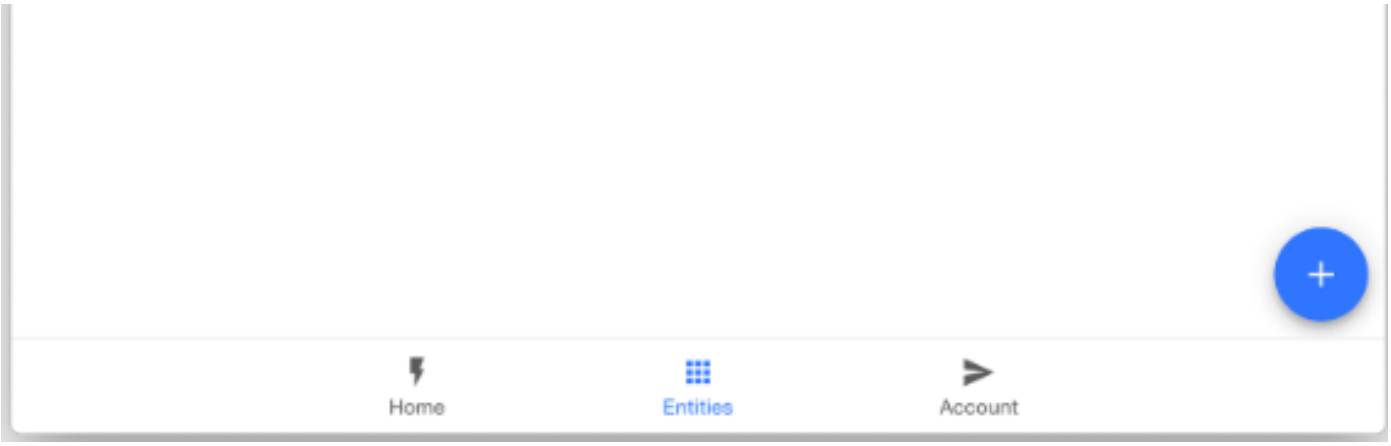
When prompted to generate this entity from an existing one, type **Y**. Enter `../app` as the path to your existing application. When prompted to regenerate entities and overwrite files, type **Y**. Enter **a** when asked about conflicting files.

Go back to your browser where your Ionic app is running (or restart it if you stopped it). Click on **Entities** on the bottom, then **Albums**. Click the blue + icon in the bottom corner, and add a new album.

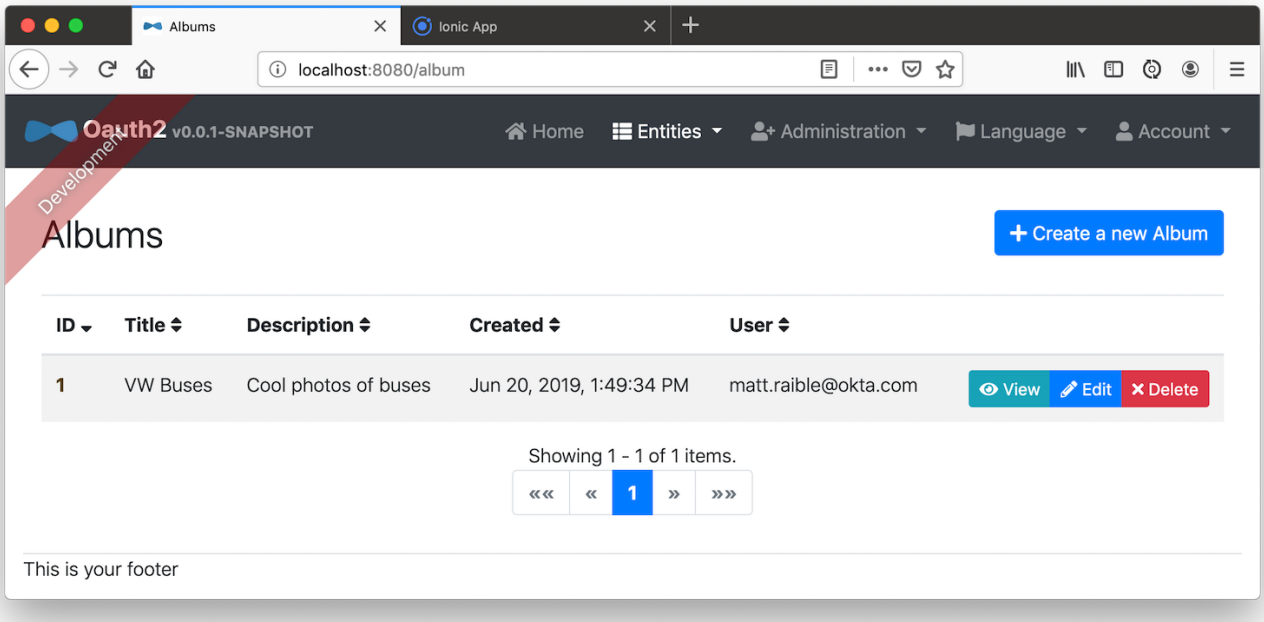


Click the checkmark in the top right corner to save your album. You'll see a success message and it listed on the next screen.





Refresh your JHipster app's album list, and you'll see it there too!



Generate code for the other entities using the following commands and the same answers as above.

```
yo jhipster-ionic:entity photo
yo jhipster-ionic:entity tag
```

Run Your Ionic App on iOS

To generate an iOS project for your Ionic application, run the following command:

```
ionic cordova prepare ios
```

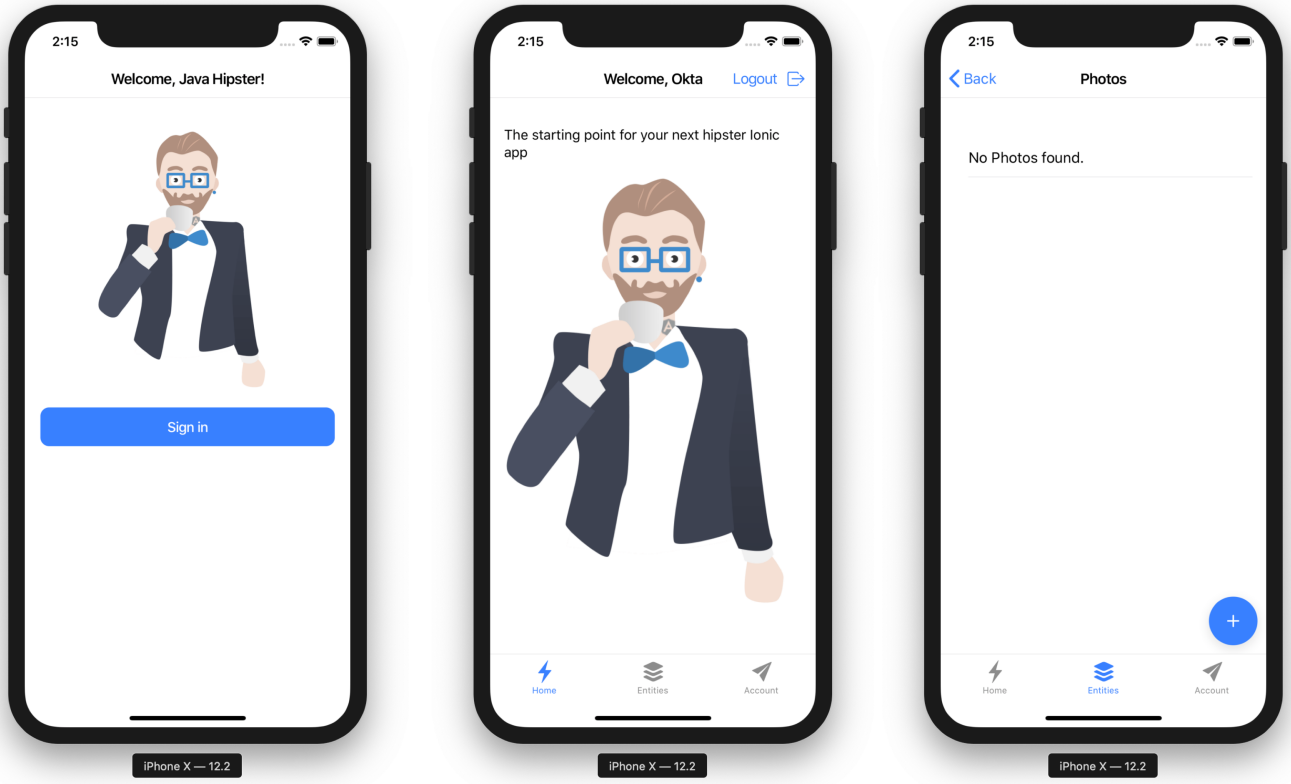
When prompted to install the `ios` platform, type **Y**. When the process completes, open your project in Xcode:

```
open platforms/ios/MyApp.xcworkspace
```

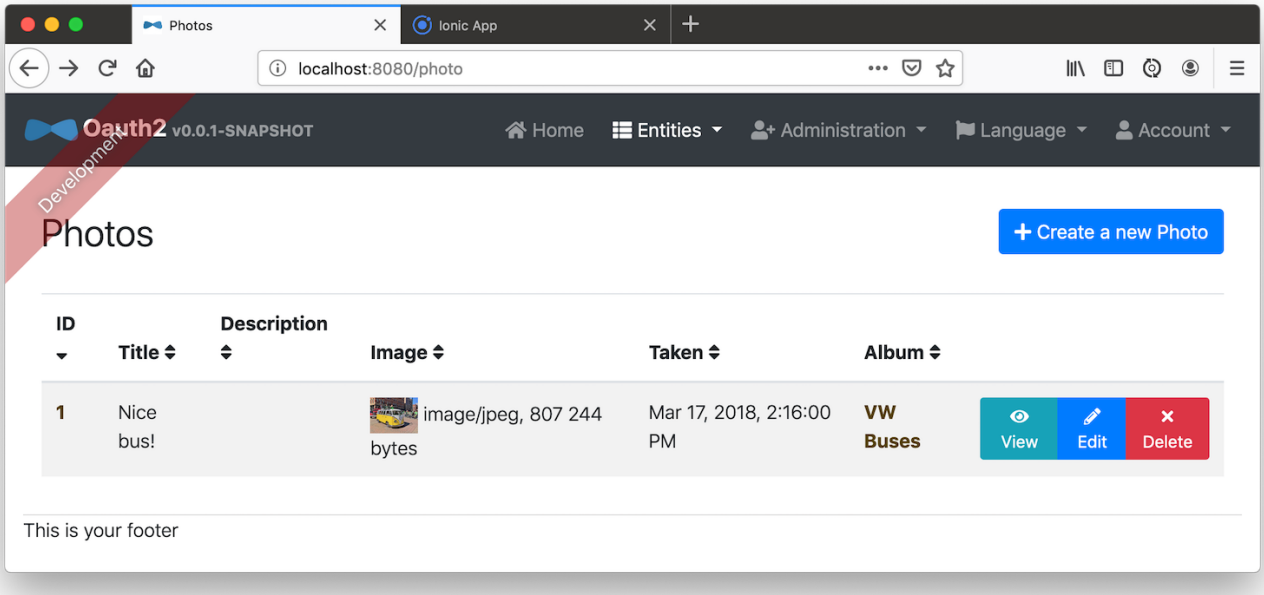
If you don't have Xcode installed, you can download it from Apple.

You'll need to configure code signing in the **General** tab, then you should be able to run your app in Simulator.

Log in to your Ionic app, tap **Entities** and view the list of photos.



Add a photo in the JHipster app at `http://localhost:8080` .



To see this new album in your Ionic app, pull down with your mouse to simulate the pull-to-refresh gesture on a phone. Looky there — it works!

There are some gestures you should know about on this screen. Clicking on the row will take you to a view screen where you can see the photo's details. You can also swipe left to expose edit and delete buttons.

Run Your Ionic App on Android

Deploying your app on Android is very similar to iOS. In short:

1. Make sure you're using Java 8
2. Run `ionic cordova prepare android`
3. Open `platforms/android` in Android Studio, upgrade Gradle if prompted
4. Set `launchMode` to `singleTask` in `AndroidManifest.xml`
5. Start your app using Android Studio
6. While your app is starting, run `adb reverse tcp:8080 tcp:8080` so the emulator can talk to JHipster

For more thorough instructions, see my Ionic 4 tutorial's Android section.

Further Reading

Secure Your Mobile App With OIDC and Ionic for JHipster

Ionic Framework: Getting Started

Learn More About Ionic 4 and JHipster 6

Ionic is a nice way to leverage your web development skills to build mobile apps. You can do most of your development in the browser, and deploy to your device when you're ready to test it. You can also just deploy your app as a PWA and not both to deploy it to an app store.

JHipster supports PWAs too, but I think Ionic apps *look* like native apps, which is a nice effect. There's a lot more I could cover about JHipster and Ionic, but this should be enough to get you started.

You can find the source code for the application developed in this post on GitHub.

I've written a few other posts on Ionic, JHipster, and Angular. Check them out if you have a moment.

- Tutorial: User Login and Registration in Ionic 4
- Java Microservices with Spring Cloud Config and JHipster
- Angular 8 + Spring Boot 2.2: Build a CRUD App Today!
- Better, Faster, Lighter Java with Java 12 and JHipster 6
- Build a Mobile App with React Native and Spring Boot

Give @oktadev a follow on Twitter if you liked this tutorial. If you have any questions, please leave a comment or post your question to Stack Overflow with a `jhipster` tag.

Build Mobile Apps with Angular, Ionic 4, and Spring Boot was originally posted on the Okta Developer Blog on June 24, 2019.

Topics: IONIC, ANGULAR, SPRING BOOT, JAVA, JHIPSTER, OIDC, OAUTH, IOS, ANDROID, TUTORIAL