

[LIVE WEBINAR] Tuesday, June 9th: Migrating to the cloud? Avoid the business risks & costs

[Sign Up Now ▶](#)

DZone > Java Zone > Build and Package a Microservices Architecture With Spring Boot and Spring Cloud

# Build and Package a Microservices Architecture With Spring Boot and Spring Cloud

by Lindsay Brunner  MVB  • Jun. 13, 19 • Java Zone • Tutorial

Secure your Java app or API service quickly and easily with user authentication and authorization libraries. Developers are free forever.

Presented by Okta

In this tutorial, I'm going to show you how to implement a microservices architecture using Spring Boot. You will also learn how to deploy the artifacts as Docker containers using Docker Compose, how to integrate authentication using Spring Profiles, and how to enable it with a production profile.

But first, let's talk about microservices.

## Understand a Modern Microservice Architecture

Microservices, as opposed to a monolith architecture, dictates you have to divide your application into small, logically-related pieces. These pieces are independent software that communicates with other pieces using HTTP or messages, for example.

There is some discussion of what size *micro* is. Some say a microservice is software that can be created in a single sprint; others say microservices can have bigger size if it is logically related (you can't mix apples and oranges, for example). I agree with Martin Fowler and think size doesn't matter that much, and it's more related to the style.

There are many advantages to microservices:

- **No high coupling risk** — Since each app lives in a different process, it is impossible to create classes that talk to each other.
- **Easy scaling** — As you already know, every service is an independent piece of software.

As such, it can be scaled up or down on demand. Moreover, since the code is *smaller* than a monolith, it probably will start up faster.

- **Multiple stacks** — You can use the best software stack for every service. No more need to use Java when, say, Python is better for what you're building.
- **Fewer merges and code conflicts** — As every service is a different repository, it is easier to handle and review commits.

However, there are some drawbacks:

- You have a new enemy — **network issues**. Is the service up? What can you do if the service is down?
- **Complex deployment process** — OK, CI/CD is here, but you now have one workflow for each service. If they use different stacks, it's possible you can't even replicate a workflow for each.
- **More complex and hard-to-understand architecture** — it depends on how you design it, but consider this: if you don't know what a method is doing, you can read its code. In a microservice architecture, this method may be in another project, and you may not even have the code.

Nowadays, it's commonly accepted that you should avoid a microservice architecture at first. After some iterations, the code division will become clearer as will the demands of your project. It is often too expensive to handle microservices until your development team starts into small projects.

## Build Microservices in Spring With Docker

You'll build two projects in this tutorial: a service (school-service) and a UI (school\_ui). The service provides the persistent layer and business logic, and the UI provides the graphical user interface. Connecting them is possible with minimal configuration.

After the initial setup, I'll talk about discovery and configuration services. Both services are an essential part of any massively distributed architecture. To prove this point, you will integrate it with OAuth 2.0 and use the configuration project to set the OAuth 2.0 keys.

Finally, each project will be transformed into a Docker image. Docker Compose will be used to simulate a *container orchestrator* as Compose will manage every container with an internal network between the services.

Lastly, Spring profiles will be introduced to change configuration based on the environment currently appropriately assigned. That way, you will have two OAuth 2.0 environments: one for development, and other for production.

Fewer words, more code! Clone this tutorial's repository and check out the `start` branch.

```
1 git clone -b start https://github.com/oktadeveloper/okta-spring-microservices-docker-example
```

The root `pom.xml` file is not a requirement. However, it can be helpful to manage multiple projects at once. Let's look inside:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema"
3     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4     <modelVersion>4.0.0</modelVersion>
5     <groupId>com.okta.developer.docker_microservices</groupId>
6     <artifactId>parent-pom</artifactId>
7     <version>0.0.1-SNAPSHOT</version>
8     <packaging>pom</packaging>
9     <name>parent-project</name>
10    <modules>
11        <module>school-service</module>
12        <module>school-ui</module>
13    </modules>
14</project>
```

This is called an *aggregate project* because it aggregates child projects. It is useful for running the same Maven task on all declared modules. The modules do not need to use the root module as a parent.

There are two modules available: a school service and a school UI.

## The School Service Microservice

The `school-service` directory contains a Spring Boot project that acts as the project's persistence layer and business rules. In a more complex scenario, you would have more services like this. The project was created using the always excellent Spring Initializr with the following configuration:

**SPRING INITIALIZR** bootstrap your application now

**Generate a** Maven Project **with** Java **and Spring Boot** 2.1.1

### Project Metadata

Artifact coordinates

**Group**

com.okta.developer.docker\_microservices

**Artifact**

### Dependencies

Add Spring Boot Starters and dependencies to your application

**Search for dependencies**

Web, Security, JPA, Actuator, Devtools...

**Selected Dependencies**

Artifact

school-service

Selected Dependencies

JPA X

Web X

Lombok X

H2 X

Generate Project alt + ↵

Don't know what to look for? Want more options? [Switch to the full version.](#)

- Group — `com.okta.developer.docker_microservices`
- Artifact — `school-service`
- Dependencies — JPA, Web, Lombok, H2

You can get more details about this project by reading `Spring Boot with PostgreSQL, Flyway, and JSONB`. To summarize, it has the entities `TeachingClass`, `Course`, `Student` and uses `TeachingClassServiceDB` and `TeachingClassController` to expose some data through a REST API. To test it, open a terminal, navigate to the `school-service` directory, and run the command below:

```
1 ./mvnw spring-boot:run
```

The application will start on port `8081` (as defined in file `school-service/src/main/resources/application.properties`), so you should be able to navigate to `http://localhost:8081` and see the returned data.

```
1> curl http://localhost:8081
2 [
3   {
4     "classId":13,
5     "teacherName":"Profesor Jirafales",
6     "teacherId":1,
7     "courseName":"Mathematics",
8     "courseId":3,
9     "numberOfStudents":2,
0     "year":1988
1   },
2   {
3     "classId":14,
4     "teacherName":"Profesor Jirafales",
5     "teacherId":1,
6     "courseName":"Spanish",
7     "courseId":4,
8     "numberOfStudents":2,
9     "year":1988
0   },
1   {
2     "classId":15,
3     "teacherName":"Professor X",
4     "teacherId":2,
5     "courseName":"Dealing with unknown",
6     "courseId":5,
```

```

7     "numberOfStudents":2,
8     "year":1995
9 },
10 {
11     "classId":16,
12     "teacherName":"Professor X",
13     "teacherId":2,
14     "courseName":"Dealing with unknown",
15     "courseId":5,
16     "numberOfStudents":1,
17     "year":1996
18 }
19 ]

```

## The Spring-Based School UI Microservice

The school UI is, as the name says, the user interface that utilizes School Service. It was created using Spring Initializr with the following options:

- Group — `com.okta.developer.docker_microservices`
- Artifact — `school-ui`
- Dependencies — Web, HATEOAS, Thymeleaf, Lombok

The UI is a single web page that lists the classes available on the database. To get the information, it connects with the `school-service` through a configuration in file `school-ui/src/main/resources/application.properties`.

```

1 service.host=localhost:8081

```

The class `SchoolController` class has all the logic to query the service:

```

1 package com.okta.developer.docker_microservices.ui.controller;
2
3 import com.okta.developer.docker_microservices.ui.dto.TeachingClassDto;
4 import org.springframework.beans.factory.annotation.*;
5 import org.springframework.core.ParameterizedTypeReference;
6 import org.springframework.http.*;
7 import org.springframework.stereotype.Controller;
8 import org.springframework.web.bind.annotation.*;
9 import org.springframework.web.client.RestTemplate;
10 import org.springframework.web.servlet.ModelAndView;
11 import java.util.List;
12
13 @Controller
14 @RequestMapping("/")
15 public class SchoolController {
16     private final RestTemplate restTemplate;

```

```

6     private final RestTemplate restTemplate;
7     private final String serviceHost;
8
9     public SchoolController(RestTemplate restTemplate, @Value("${service.host}") String serviceHost) {
10         this.restTemplate = restTemplate;
11         this.serviceHost = serviceHost;
12     }
13
14     @RequestMapping("")
15     public ModelAndView index() {
16         return new ModelAndView("index");
17     }
18
19     @GetMapping("/classes")
20     public ResponseEntity<List<TeachingClassDto>> listClasses(){
21         return restTemplate
22             .exchange("http://" + serviceHost + "/class", HttpMethod.GET, null,
23                 new ParameterizedTypeReference<List<TeachingClassDto>>() {});
24     }
25 }

```

As you can see, there is a hard-coded location for the service. You can change the property setting with an environment variable like this `-Dservice.host=localhost:9090`. Still, it has to be manually defined. How about having many instances of *school-service* application? Impossible at the current stage.

With *school-service* turned on, start `school-ui`, and navigate to it in a browser at `http://localhost:8080`:

```
1 ./mvnw spring-boot:run
```

You should see a page like the following:

## School classes

Course	Teacher	Year	Number of studends
Mathematics	Profesor Jirafales	1988	2
Spanish	Profesor Jirafales	1988	2
Dealing with unknown	Professor X	1995	2
Dealing with unknown	Professor X	1996	1

## Build a Discovery Server With Spring Cloud and Eureka

Now, you have a working application that uses two services to provide the information to end-user. What is wrong with it? In modern applications, developers (or operations) usually don't

user. What is wrong with it? In modern applications, developers (or operations) usually don't know where or what port an application might be deployed on. The deployment should be automated so that no one *cares* about server names and physical location. (Unless you work inside a data center. If you do, I hope you care!)

Nonetheless, it is essential to have a tool that helps the services to discover their counterparts. There are many solutions available, and for this tutorial, we are going to use *Eureka* from Netflix as it has outstanding Spring support.

Go back to [start.spring.io](http://start.spring.io) and create a new project as follows:

- Group: `com.okta.developer.docker_microservices`
- Artifact: `discovery`
- Dependencies: Eureka Server

Edit the main `DiscoveryApplication.java` class to add an `@EnableEurekaServer` annotation:

```
1 package com.okta.developer.docker_microservices.discovery;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
6
7 @SpringBootApplication
8 @EnableEurekaServer
9 public class DiscoveryApplication {
10     public static void main(String[] args) {
11         SpringApplication.run(DiscoveryApplication.class, args);
12     }
13 }
```

And, you'll need to update its `application.properties` file so it runs on port 8761 and doesn't try to register with itself.

```
1 spring.application.name=discovery-server
2 server.port=8761
3 eureka.client.register-with-eureka=false
4 eureka.client.fetch-registry=false
```

Let's define each property:

- `spring.application.name` — The name of the application, also used by the discovery service to *discover* a service. You'll see that every other application has an

application name too.

- `server.port` — The port the server is running. `8761` is the default port for Eureka server.
- `eureka.client.register-with-eureka` — Tells Spring not to register itself into the discovery service.
- `eureka.client.fetch-registry` — Indicates this instance should not fetch discovery information from the server.

Now, run and access `http://localhost:8761`.

```
1 ./mvnw spring-boot:run
```

The screenshot shows the Spring Eureka web interface. At the top, there's a navigation bar with the Spring Eureka logo and links for HOME and LAST 1000 SINCE STARTUP. Below this, the 'System Status' section displays two tables. The left table shows 'Environment' as 'test' and 'Data center' as 'default'. The right table shows 'Current time' as '2018-12-07T23:34:00 -0200', 'Uptime' as '00:01', 'Lease expiration enabled' as 'false', 'Renews threshold' as '1', and 'Renews (last min)' as '0'. Below the system status, the 'DS Replicas' section shows 'localhost'. The 'Instances currently registered with Eureka' section shows a table with columns 'Application', 'AMIs', 'Availability Zones', and 'Status'. The table is empty, with the text 'No instances available' displayed below it.

The screen above shows the Eureka server ready to register new services. Now, it is time to change *school-service* and *school-ui* to use it.

**NOTE:** If you receive a `ClassNotFoundException: javax.xml.bind.JAXBContext` error on startup, it's because you're running on Java 11. You can add JAXB dependencies to your `pom.xml` to fix this.

```
1 <dependency>
2   <groupId>javax.xml.bind</groupId>
3   <artifactId>jaxb-api</artifactId>
4   <version>2.3.1</version>
5 </dependency>
6 <dependency>
7   <groupId>org.glassfish.jaxb</groupId>
8   <artifactId>jaxb-runtime</artifactId>
9   <version>2.3.2</version>
10</dependency>
```



# Use Service Discovery to Communicate Between Microservices

First, it is important to add the required dependencies. Add the following to both `pom.xml` file (in the *school-service* and *school-ui* projects):

```
1 <dependency>
2     <groupId>org.springframework.cloud</groupId>
3     <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
4 </dependency>
```

This module is part of the Spring Cloud initiative and, as such, needs a new dependency management node as follows (don't forget to add to both projects):

```
1 <dependencyManagement>
2     <dependencies>
3         <dependency>
4             <groupId>org.springframework.cloud</groupId>
5             <artifactId>spring-cloud-dependencies</artifactId>
6             <version>${spring-cloud.version}</version>
7             <type>pom</type>
8             <scope>import</scope>
9         </dependency>
10    </dependencies>
11 </dependencyManagement>
```

Now, you need to configure both applications to register with Eureka.

In the `application.properties` file of both projects, add the following lines:

```
1 eureka.client.serviceUrl.defaultZone=${EUREKA_SERVER:http://localhost:8761/eureka}
2 spring.application.name=school-service
```

Don't forget to change the application name from `school-service` to `school-ui` in the *school-ui* project. Notice there is a new kind of parameter in the first line:

`{EUREKA_SERVER:http://localhost:8761/eureka}`. It means "if environment variable `EUREKA_SERVER` exists, use its value, if not, here's a default value." This will be useful in future steps.

You know what? Both applications are ready to register themselves into the discovery service. You don't need to do anything more. Our primary objective is that *school-ui* project does not need to know *where* *school-service* is. As such, you need to change `SchoolController` (in

need to know *where* *school-service* is. As such, you need to change `SchoolController` (in the `school-ui` project) to use `school-service` in its REST endpoint. You can also remove the `serviceHost` variable in this class.

```
1 package com.okta.developer.docker_microservices.ui.controller;
2
3 import com.okta.developer.docker_microservices.ui.dto.TeachingClassDto;
4 import org.springframework.core.ParameterizedTypeReference;
5 import org.springframework.http.HttpMethod;
6 import org.springframework.http.ResponseEntity;
7 import org.springframework.stereotype.Controller;
8 import org.springframework.web.bind.annotation.GetMapping;
9 import org.springframework.web.bind.annotation.RequestMapping;
10 import org.springframework.web.client.RestTemplate;
11 import org.springframework.web.servlet.ModelAndView;
12
13 import java.util.List;
14
15 @Controller
16 @RequestMapping("/")
17 public class SchoolController {
18     private final RestTemplate restTemplate;
19
20     public SchoolController(RestTemplate restTemplate) {
21         this.restTemplate = restTemplate;
22     }
23
24     @RequestMapping("")
25     public ModelAndView index() {
26         return new ModelAndView("index");
27     }
28
29     @GetMapping("/classes")
30     public ResponseEntity<List<TeachingClassDto>> listClasses() {
31         return restTemplate
32             .exchange("http://school-service/classes", HttpMethod.GET, null,
33                 new ParameterizedTypeReference<List<TeachingClassDto>>() {});
34     }
35 }
```

Before integrating Eureka, you had a configuration pointing out where *school-service* was. Now, you've changed the service calls to use the name used by the other service: no ports, no hostname. The service you need is somewhere, and you don't need to know where.

The *school-service* may have multiple instances of and it would be a good idea to load balance the calls between the instances. Thankfully, Spring has a simple solution: on the `RestTemplate` bean creation, add `@LoadBalanced` annotation as follows. Spring will manage multiple instance calls each time you ask something to the server.

```
1 package com.okta.developer.docker_microservices.ui;
2
3 import org.springframework.boot.SpringApplication;
```

```

3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.cloud.client.loadbalancer.LoadBalanced;
6 import org.springframework.context.annotation.Bean;
7 import org.springframework.web.client.RestTemplate;
8 import org.springframework.web.servlet.config.annotation.*;
9
10 @SpringBootApplication
11 public class UIWebApplication implements WebMvcConfigurer {
12
13     public static void main(String[] args) {
14         SpringApplication.run(UIWebApplication.class, args);
15     }
16
17     @Bean
18     @LoadBalanced
19     public RestTemplate restTemplate() {
20         return new RestTemplate();
21     }
22
23     @Override
24     public void addResourceHandlers(ResourceHandlerRegistry registry) {
25         if(!registry.hasMappingForPattern("/static/**")) {
26             registry.addResourceHandler("/static/**")
27                 .addResourceLocations("classpath:/static/", "classpath:/static/js/");
28         }
29     }
30 }

```

Now, start restart *school-service* and *school-ui* (and keep the Discovery service up). Have a quick look at <http://localhost:8761> again:

#### Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
SCHOOL-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.16:school-service:8081
SCHOOL-UI	n/a (1)	(1)	UP (1) - 192.168.0.16:school-ui

Now, your services are sharing info with the Discovery server. You can test the application again and see that it work as always. Just go to <http://localhost:8080> in your favorite browser.

## Add a Configuration Server to Your Microservices Architecture

While this configuration works, it's even better to remove any trace of configuration values in the project's source code. First, the configuration URL was removed from the project and became managed by a service. Now, you can do a similar thing for every configuration on the project using Spring Cloud Config.

First, create the configuration project using Spring Initializr and the following parameters:

First, create the configuration project using Spring Initializr and the following parameters.

- Group: `com.okta.developer.docker_microservices`
- Artifact: `config`
- Dependencies: Config Server, Eureka Discovery

In the main class, add `@EnableConfigServer`:

```
1 package com.okta.developer.docker_microservices.config;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.cloud.config.server.EnableConfigServer;
6
7 @SpringBootApplication
8 @EnableConfigServer
9 public class ConfigApplication {
10     ...
11 }
```

Add the following properties and values in the project's `application.properties`:

```
1 spring.application.name=CONFIGSERVER
2 server.port=8888
3 spring.profiles.active=native
4 spring.cloud.config.server.native.searchLocations=.
5 eureka.client.serviceUrl.defaultZone=${EUREKA_SERVER:http://localhost:8761/eureka}
```

Some explanation about the properties:

- `spring.profiles.active=native` — Indicates Spring Cloud Config must use the native file system to obtain the configuration. Normally, Git repositories are used, but we are going to stick with native filesystem for simplicity sake.
- `spring.cloud.config.server.native.searchLocations` — The path containing the configuration files. If you change this to a specific folder on your hard drive, make sure and create the `school-ui.properties` file in it.

Now, you need something to configure and apply to this example. How about Okta's configuration? Let's put our *school-ui* behind an authorization layer and use the property values provided by the configuration project.

You can register for a free-forever developer account that will enable you to create as many user and applications you need to use! After creating your account, create a new Web

## Application in Okta's dashboard (**Applications > Add Application**):

### Create New Application


1


Platform


2


Settings

An application in Okta represents an integration with the software you're building. Choose your platform, and we'll recommend settings on the next step.

  
Native  
iOS, Android

  
Single-Page App  
Angular, React, etc.

  
Web  
.NET, Java, etc.

  
Service  
Machine-to-Machine

And fill the next form with the following values:

All these settings can be changed at any time.

APPLICATION SETTINGS

Name

okta-docker

Base URIs  
Optional

http://localhost:8080/

+ Add URI

The domains where your application runs. Trusted Origins will be created for these URIs, and will be the only domains Okta accepts API calls from. [Docs](#)

Login redirect URIs

http://localhost:8080/authorization-code/callback

+ Add URI

Okta will send OAuth authorization response to these URIs. Add your application's callback endpoint. [Docs](#)

Group assignments  
Optional

Everyone

Users can only sign in to apps that they are assigned to. Group assignments are easier to manage than individual users.

Grant type allowed

Client acting on behalf of itself

☐ Client Credentials

Client acting on behalf of a user

☒ Authorization Code

☐ Refresh Token

☐ Implicit (Hybrid)

Okta can authorize your native app's requests with these OAuth 2.0 grant types. Limit the allowed grant types to minimize security risks [Docs](#)

Previous

Cancel

Done

The page will return you an application ID and a secret key. Keep them safe and create a file called `school-ui.properties` in the root folder of the `config` project with the following contents. Do not forget to populate the variable values:

```
1 okta.oauth2.issuer=https://okta.okta.com/oauth2/default
2 okta.oauth2.clientId={yourClientId}
3 okta.oauth2.clientSecret={yourClientSecret}
```

Now, run the `config` project and check if it's getting the configuration data properly:

```
1 ./mvnw spring-boot:run
2 > curl http://localhost:8888/school-ui.properties
3 okta.oauth2.clientId: YOUR_CLIENT_ID
4 okta.oauth2.clientSecret: YOUR_CLIENT_SECRET
5 okta.oauth2.issuer: https://YOUR_DOMAIN/oauth2/default
```

## Change School UI to Use Spring Cloud Config and OAuth 2.0

Now, you need to change the Spring UI project a little bit.

First, you need to change `school-ui/pom.xml` and add some new dependencies:

```
1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-starter-config</artifactId>
4 </dependency>
5 <dependency>
6   <groupId>com.okta.spring</groupId>
7   <artifactId>okta-spring-boot-starter</artifactId>
8   <version>1.1.0</version>
9 </dependency>
10 <dependency>
11   <groupId>org.thymeleaf.extras</groupId>
12   <artifactId>thymeleaf-extras-springsecurity5</artifactId>
13 </dependency>
```

Create a new `SecurityConfiguration` class in the `com.okta...ui.config` package:

```
1 package com.okta.developer.docker_microservices.ui;
2
3 import org.springframework.context.annotation.Configuration;
4 import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;
5 import org.springframework.security.config.annotation.web.builders.HttpSecurity;
6 import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
7
8 @Configuration
9 @EnableGlobalMethodSecurity(prePostEnabled = true)
10 public class SpringSecurityConfiguration extends WebSecurityConfigurerAdapter {
11
12     @Override
13     protected void configure(HttpSecurity http) throws Exception {
14         http
15             .authorizeRequests()
16                 .antMatchers("/").permitAll()
17                 .anyRequest().authenticated()
18             .and()
19                 .logout().logoutSuccessUrl("/")
20             .and()
21                 .oauth2Login();
22     }
23 }
```

Change your `SchoolController` so only users with `scope profile` will be allowed (every authenticated user will have it).

```
1 import org.springframework.security.access.prepost.PreAuthorize;
2
3 ....
4
5 @GetMapping("/classes")
6 @PreAuthorize("hasAuthority('SCOPE_profile')")
7 public ResponseEntity<List<TeachingClassDto>> listClasses(){
8     return restTemplate
9         .exchange("http://school-service/class", HttpMethod.GET, null,
10             new ParameterizedTypeReference<List<TeachingClassDto>>() {});
11 }
```

Some configurations need to be defined at project boot time. Spring had a clever solution to locate properly and extract configuration data *before* context startup. You need to create a file `src/main/resources/bootstrap.yml` like this:

```
1 spring:
2     cloud:
3         config:
4             uri: http://localhost:8888
```

```

1 eureka:
2   client:
3     serviceUrl:
4       defaultZone: ${EUREKA_SERVER:http://localhost:8761/eureka}
5 spring:
6   application:
7     name: school-ui
8   cloud:
9     config:
10      discovery:
11        enabled: true
12        service-id: CONFIGSERVER

```

The bootstrap file creates a pre-boot Spring Application Context responsible for extracting configuration before the real application starts. You need to move all properties from `application.properties` to this file as Spring needs to know where your Eureka Server is located and how it should search for configuration. In the example above, you enabled configuration over discovery service ( `spring.cloud.config.discovery.enabled` ) and specified the Configuration `service-id`.

Change your `application.properties` file so it only has one OAuth 2.0 property:

```

1 okta.oauth2.redirect-uri=/authorization-code/callback

```

The last file to modify is `src/main/resources/templates/index.html`. Adjust it to show a login button if the user is not authenticated, and a logout button if the user is logged in.

```

1 <!doctype html>
2 <html lang="en" xmlns:th="http://www.thymeleaf.org">
3 <head>
4   <!-- Required meta tags -->
5   <meta charset="utf-8">
6   <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
7
8   <!-- Bootstrap CSS -->
9   <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/boot
0
1   <title>Hello, world!</title>
2 </head>
3 <body>
4 <nav class="navbar navbar-default">
5   <form method="post" th:action="@{/logout}" th:if="${#authorization.expression('isAuthent
6     <input type="hidden" th:name="${_csrf.parameterName}" th:value="${_csrf.token}" />
7     <button id="logout-button" type="submit" class="btn btn-danger">Logout</button>
8   </form>
9   <form method="get" th:action="@{/oauth2/authorization/okta}" th:unless="${#authorization
0     <button id="login-button" class="btn btn-primary" type="submit">Login</button>
1   </form>

```



```

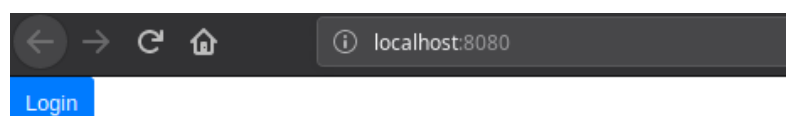
2 </nav>
3
4 <div id="content" th:if="${#authorization.expression('isAuthenticated()')}">
5     <h1>School classes</h1>
6
7     <table id="classes">
8         <thead>
9             <tr>
10                 <th>Course</th>
11                 <th>Teacher</th>
12                 <th>Year</th>
13                 <th>Number of students</th>
14             </tr>
15         </thead>
16         <tbody>
17
18         </tbody>
19     </table>
20
21     <!-- Optional JavaScript -->
22     <!-- jQuery first, then Popper.js, then Bootstrap JS -->
23
24     <script src="https://code.jquery.com/jquery-3.3.1.min.js" crossorigin="anonymous"></script>
25     <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js"
26     <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js" int
27     <script src="static/js/school_classes.js"></script>
28 </div>
29
30 </body>
31 </html>

```

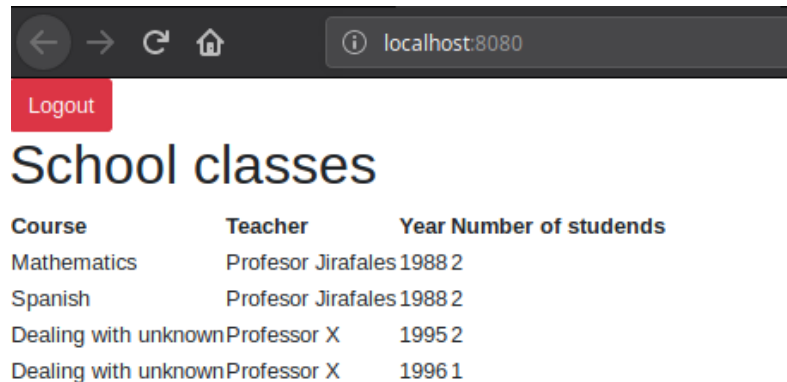
There are some Thymeleaf properties you should know about in this HTML:

- `@{/logout}` — returns the logout URL defined on the back-end
- `th:if="${#authorization.expression('isAuthenticated()')}"` — only print the HTML if the user is **logged in**
- `@{/oauth2/authorization/okta}` — this is the URL that Spring Security redirects to for Okta. You could link to `/login` as well, but that just renders the same link and you have to click on it.
- `th:unless="${#authorization.expression('isAuthenticated()')}"` - only print the HTML inside the node if the user is **logged off**

Now restart the configuration project and school-ui again. If you navigate to typing `http://localhost:8080`, you should see the following screen:



After logged in, the screen should appear like this one:



Congratulations, you created a microservices architecture using Spring Cloud config and Eureka for service discovery! Now, let's go one step further and Dockerize every service.

## Use Docker to Package Your Spring Apps

Docker is a marvelous technology that allows creating system images similar to *Virtual Machines* but that shares the same Kernel of the host operating system. This feature increases system performance and startup time. Also, Docker provided an ingenious built system that guarantees once an image is created; it won't be changed, ever. In other words: no more "it works on my machine!"

**TIP:** Need a deeper Docker background? Have a look at our Developer's Guide to Docker.

You'll need to create one Docker image for each project. Each image should have the same Maven configuration and `Dockerfile` content in the root folder of each project (e.g. `school-ui/Dockerfile`).

In each project's pom, add the `dockerfile-maven-plugin`:

```
1 <plugins>
2   ...
3   <plugin>
4     <groupId>com.spotify</groupId>
5     <artifactId>dockerfile-maven-plugin</artifactId>
6     <version>1.4.9</version>
7     <executions>
8       <execution>
9         <id>default</id>
10        <goals>
11          <goal>build</goal>
12          <goal>push</goal>
13        </goals>
14      </execution>
```

```

5     </executions>
6     <configuration>
7         <repository>developer.okta.com/microservice-docker-${project.artifactId}</repository>
8         <tag>${project.version}</tag>
9         <buildArgs>
10             <JAR_FILE>${project.build.finalName}.jar</JAR_FILE>
11         </buildArgs>
12     </configuration>
13 </plugin>
14</plugins>

```

This XML configures the Dockerfile Maven plugin to build a Docker image every time you run `./mvnw install`. Each image will be created with the name `developer.okta.com/microservice-docker-${project.artifactId}` where `project.artifactId` varies by project.

Create a `Dockerfile` file in the root directory of each project.

```

1 FROM openjdk:8-jdk-alpine
2 VOLUME /tmp
3 ADD target/*.jar app.jar
4 ENV JAVA_OPTS=""
5 ENTRYPOINT [ "sh", "-c", "java $JAVA_OPTS -Djava.security.egd=file:/dev/./urandom -jar /app.jar" ]

```

The `Dockerfile` follows what is recommended by Spring Boot with Docker.

Now, change `school-ui/src/main/resources/bootstrap.yml` to add a new `failFast` setting:

```

1 eureka:
2   client:
3     serviceUrl:
4       defaultZone: ${EUREKA_SERVER:http://localhost:8761/eureka}
5 spring:
6   application:
7     name: school-ui
8   cloud:
9     config:
10      discovery:
11        enabled: true
12        serviceId: CONFIGSERVER
13      failFast: true

```

The `spring.cloud.failFast: true` setting tells Spring Cloud Config to terminate the application as soon as it can't find the configuration server. This will be useful for the next

application as soon as it can find the configuration server. This will be useful for the next step.

## Add Docker Compose to Run Everything

Create a new file called `docker-compose.yml` that defines how each project starts:

```
1 version: '3'
2 services:
3   discovery:
4     image: developer.okta.com/microservice-docker-discovery:0.0.1-SNAPSHOT
5     ports:
6       - 8761:8761
7   config:
8     image: developer.okta.com/microservice-docker-config:0.0.1-SNAPSHOT
9     volumes:
10      - ./config-data:/var/config-data
11     environment:
12       - JAVA_OPTS=
13         -DEUREKA_SERVER=http://discovery:8761/eureka
14         -Dspring.cloud.config.server.native.searchLocations=/var/config-data
15     depends_on:
16       - discovery
17     ports:
18       - 8888:8888
19   school-service:
20     image: developer.okta.com/microservice-docker-school-service:0.0.1-SNAPSHOT
21     environment:
22       - JAVA_OPTS=
23         -DEUREKA_SERVER=http://discovery:8761/eureka
24     depends_on:
25       - discovery
26       - config
27   school-ui:
28     image: developer.okta.com/microservice-docker-school-ui:0.0.1-SNAPSHOT
29     environment:
30       - JAVA_OPTS=
31         -DEUREKA_SERVER=http://discovery:8761/eureka
32     restart: on-failure
33     depends_on:
34       - discovery
35       - config
36     ports:
37       - 8080:8080
```

As you can see, each project is now a declared service in Docker compose the file. It'll have its ports exposed and some other properties.

- All projects besides *discovery* will have a variable value `-DEUREKA_SERVER=http://discovery:8761/eureka`. This will tell where to find the Discovery server. Docker Compose creates a virtual network between the services and the DNS name used for each service is its name: that's why it's possible to use

discovery as the hostname.

- The Config service will have a volume going to configuration files. This volume will be mapped to `/var/config-data` inside the docker container. Also, the property `spring.cloud.config.server.native.searchLocations` will be overwritten to the same value. You must store the file `school-ui.properties` in the same folder specified on the volume mapping (in the example above, the *relative* folder `./config-data`).
- The *school-ui* project will have the property `restart: on-failure`. This set Docker Compose to restart the application as soon as it fails. Using together with `failFast` property allows the application to keep trying to start until the *Discovery* and *Config* projects are completely ready.

And that's it! Now, build the images:

```
1 cd config && ./mvnw clean install
2 cd ../discovery && ./mvnw clean install
3 cd .. && ./mvnw clean install
```

The last command will likely fail with the following error in the `school-ui` project:

```
1 java.lang.IllegalStateException: Failed to load ApplicationContext
2 Caused by: java.lang.IllegalStateException: No instances found of configserver (CONFIGSERVER)
```

To fix this, create a `school-ui/src/test/resources/test.properties` file and add properties so Okta's config passes, and it doesn't use discovery or the config server when testing.

```
1 okta.oauth2.issuer=https://okta.okta.com/oauth2/default
2 okta.oauth2.clientId=TEST
3 spring.cloud.discovery.enabled=false
4 spring.cloud.config.discovery.enabled = false
5 spring.cloud.config.enabled = false
```

Then, modify `UIWebApplicationTests.java` to load this file for test properties:

```
1 import org.springframework.test.context.TestPropertySource;
2
3 ...
4 @TestPropertySource(locations="classpath:test.properties")
```

```
4 testPropertySource(locations=classpath:test.properties )
5 public class UIWebApplicationTests {
6     ...
7 }
```

Now, you should be able to run `./mvnw clean install` in the `school-ui` project.

Once that completes, run Docker Compose to start all your containers (in the same directory where `docker-compose.yml` is).

```
1 docker-compose up -d
2 Starting okta-microservice-docker-post-final_discovery_1 ... done
3 Starting okta-microservice-docker-post-final_config_1    ... done
4 Starting okta-microservice-docker-post-final_school-ui_1 ... done
5 Starting okta-microservice-docker-post-final_school-service_1 ... done
```

Now, you should be able to browse the application as you did previously.

## Use Spring Profiles to Modify Your Microservices' Configuration

Now you've reached the last stage of today's journey through microservices. Spring Profiles is a powerful tool. Using profiles, it is possible to modify program behavior by injecting different dependencies or configurations completely.

Imagine you have a well-architected software that has its persistence layer separated from business logic. You also provide support for MySQL and PostgreSQL, for example. It is possible to have different data access classes for each database that will be only loaded by the defined profile.

Another use case is for configuration: different profiles might have different configurations. Take authentication, for instance. Will your test environment have authentication? If it does, it shouldn't use the same user directory as production.

Change your configuration project to have two apps in Okta: one default (for development) and another for production. Create a new Web application on Okta website and name it "okta-docker-production."

Now, in your `config` project, create a new file called `school-ui-production.properties`. You already have `school-ui.properties`, which will be used by every School UI instance. When adding the environment at the end of the file, Spring will merge both files and take precedence over the most specific file. Save the

file with your production app's client ID and secret, like this:

### **school-ui-production.properties**

```
1 okta.oauth2.clientId={YOUR_PRODUCTION_CLIENT_ID}
2 okta.oauth2.clientSecret={YOUR_PRODUCTION_CLIENT_SECRET}
```

Now, run the configuration project using Maven, then run the following two `curl` commands:

```
1 ./mvnw spring-boot:run
2
3 > curl http://localhost:8888/school-ui.properties
4
5 okta.oauth2.issuer: https://{yourOktaDomain}/oauth2/default
6 okta.oauth2.clientId: ==YOUR DEV CLIENT ID HERE==
7 okta.oauth2.clientSecret: ==YOUR DEV CLIENT SECRET HERE==
8
9 > curl http://localhost:8888/school-ui-production.properties
0 okta.oauth2.issuer: https://{yourOktaDomain}/oauth2/default
1 okta.oauth2.clientId: ==YOUR PROD CLIENT ID HERE==
2 okta.oauth2.clientSecret: ==YOUR PROD CLIENT SECRET HERE==
```

As you can see, even though the file `school-ui-production` has two properties, the `config` project displays three (since the configurations are merged).

Now, you can change the `school-ui` service in the `docker-compose.yml` to use the `production` profile:

```
1 school-ui:
2   image: developer.okta.com/microservice-docker-school-ui:0.0.1-SNAPSHOT
3   environment:
4     - JAVA_OPTS=
5       -DEUREKA_SERVER=http://discovery:8761/eureka
6       -Dspring.profiles.active=production
7   restart: on-failure
8   depends_on:
9     - discovery
0     - config
1   ports:
2     - 8080:8080
```

You'll also need to copy `school-ui-production.properties` to your `config-data` directory. Then, shut down all your Docker containers and restart them.

```
1 docker-compose down
2 docker-compose up -d
```

You should see the following printed in the logs of the `school-ui` container:

```
1 The following profiles are active: production
```

That's it! Now, you have your microservices architecture running with a production profile. Huzzah!

**TIP:** If you want to prove your `okta-docker-production` app is used and not `okta-docker`, you can deactivate the `okta-docker` app in Okta and confirm you can still log in at `http://localhost:8080`.

## Learn More About Microservices, Spring, Docker, and Modern Application Security

In this post, you learned more about microservices and how to deploy them, along with:

- What is a microservice?
- How services should discover their dependencies without previously knowing where they are located.
- How to maintain a distributed configuration with a central point of information. The configuration can manage one or more applications and environments.
- How to configure OAuth 2.0 using Spring Cloud Config.
- How to deploy microservices using Docker and Docker Compose.
- How to use Spring Profiles to deploy in a production environment.

You can find the completed source code for this tutorial on GitHub at [oktadeveloper/okta-spring-microservices-docker-example](https://github.com/oktadeveloper/okta-spring-microservices-docker-example).

If you're interested in learning more about microservices, or modern application development in Spring, I encourage you to check out these resources:

- Build and Secure Microservices with Spring Boot 2.0 and OAuth 2.0
- Developer a Microservices Architecture with JHipster and OAuth 2.0
- Build a Microservices Architecture for Microbrews with Spring Boot



- Spring Boot 2.1: Outstanding OIDC, OAuth 2.0, and Reactive API Support
- Build a Reactive App with Spring Boot and MongoDB

If you have any questions about this post, please leave a comment below. You can follow @oktadev on Twitter for more awesome content!

Build Spring Microservices and Dockerize Them for Production was originally published to the Okta developer blog on February 28, 2019.

---

Secure your Java app or API service quickly and easily with user authentication and authorization libraries. Developers are free forever. [Try Okta.](#)

Presented by Okta

---

## Like This Article? Read More From DZone



**DZone Article**  
**Java Microservices: Code Examples, Tutorials, and More**



**DZone Article**  
**Microservices Using Spring Boot and Spring Cloud – Part 1: Overview**



**DZone Article**  
**This Week in Spring: Spring Cloud, Microservices, and More**



**Free DZone Refcard**  
**Java 14**

Topics: DOCKER , JAVA , MICROSERVICE , SPRING BOOT , SPRING CLOUD , SPRING FRAMEWORK , SPRING SECURITY , TUTORIAL

Published at DZone with permission of Lindsay Brunner , DZone MVB. [See the original article here.](#)

Opinions expressed by DZone contributors are their own.