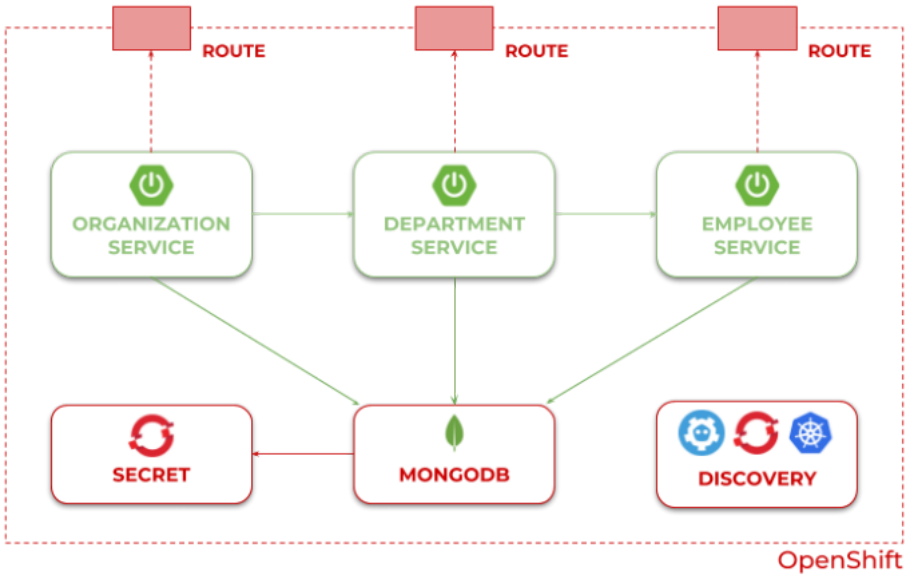
Running Java Microservices on OpenShift using Source-2-Image

Running Java Microservices on OpenShift using Source-2-Image

One of the reasons you would prefer OpenShift instead of Kubernetes is the simplicity of running new applications. When working with plain Kubernetes you need to provide an already built image together with the set of descriptor templates used for deploying it. OpenShift introduces **Source-2-Image** feature used for building reproducible Docker images from application source code. With S2I you don't have to provide any Kubernetes YAML templates or build a Docker image by yourself, OpenShift will do it for you. Let's see how it works. The best way to test it locally is via Minishift. But the first step is to prepare sample applications source code.

1. Prepare application code

I have already described how to run your Java applications on Kubernetes in one of my previous articles Quick Guide to Microservices with Kubernetes, Spring Boot 2.0 and Docker. We will use the same source code as used in that article now, so you would be able to compare those two different approaches. Our source code is available on GitHub in repository **sample-spring-microservices-new**. We will modify a little the version used in Kubernetes by removing Spring Cloud Kubernetes library and including some additional resources. The current version is available in the branch openshift.
Our sample system consists of three microservices which communicate with each other and use Mongo database backend. Here's the diagram that illustrates our architecture.



Every microservice is a Spring Boot application, which uses Maven as a built tool. After including spring-boot-maven-plugin it is able to generate single fat jar with all dependencies, which is required by source-2-image builder.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

Every application includes starters for Spring Web, Spring Actuator and Spring Data MongoDB for integration with Mongo database. We will also include libraries for generating Swagger API documentation, and Spring Cloud OpenFeign for these applications which call REST endpoints exposed by other microservices.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
  <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2>/version<
  </dependency>
  <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
  </dependency>
</dependencies>
```

Every Spring Boot application exposes REST API for simple CRUD operations on a given resource. The Spring Data repository bean is injected into the controller.

```
@RestController
@RequestMapping("/employee")
public class EmployeeController {

  private static final Logger LOGGER = LoggerFactory.getLogger(EmployeeController.class);

  @Autowired
  EmployeeRepository repository;

  @PostMapping("/")
  public Employee add(@RequestBody Employee employee) {
    LOGGER.info("Employee add: {}", employee);
    return repository.save(employee);
  }
```

```
  @GetMapping("/{id}")
  public Employee findById(@PathVariable("id") String id) {
    LOGGER.info("Employee find: id={}", id);
    return repository.findById(id).get();
  }

  @GetMapping("/")
  public Iterable<Employee> findAll() {
    LOGGER.info("Employee find");
    return repository.findAll();
  }

  @GetMapping("/department/{departmentId}")
  public List<Employee> findByDepartment(@PathVariable("departmentId") Long departmentId) {
    LOGGER.info("Employee find: departmentId={}", departmentId);
    return repository.findByDepartmentId(departmentId);
  }

  @GetMapping("/organization/{organizationId}")
  public List<Employee> findByOrganization(@PathVariable("organizationId") Long organizationId) {
    LOGGER.info("Employee find: organizationId={}", organizationId);
    return repository.findByOrganizationId(organizationId);
  }

}
```

The application expects to have environment variables pointing to the database name, user and password.

```
spring:
  application:
    name: employee
  data:
    mongodb:
      uri: mongodb://${MONGO_DATABASE_USER}:${MONGO_DATABASE_PASSWORD}@mongodb/${MONGO_DATABASE_NAME}
```

Inter-service communication is realized through OpenFeign – a declarative REST client. It is included in department and organization microservices.

```
@FeignClient(name = "employee", url = "${microservices.employee.url}")
public interface EmployeeClient {

  @GetMapping("/employee/organization/{organizationId}")
  List<Employee> findByOrganization(@PathVariable("organizationId") String organizationId);

}
```

The address of the target service accessed by Feign client is set inside application.yml. The communication is realized via OpenShift/Kubernetes services. The name of each service is also injected through an environment variable.

```
spring:
  application:
    name: organization
  data:
    mongodb:
      uri: mongodb://${MONGO_DATABASE_USER}:${MONGO_DATABASE_PASSWORD}@mongodb/${MONGO_DATABASE_NAME}
microservices:
  employee:
    url: http://${EMPLOYEE_SERVICE}:8080
  department:
    url: http://${DEPARTMENT_SERVICE}:8080
```

2. Running Minishift

To run Minishift locally you just have to download it from that site, copy minishift.exe (for Windows) to your PATH directory and start using minishift start command. For more details you may refer to my previous article about OpenShift and Java applications Quick guide to deploying Java apps on OpenShift. The current version of Minishift used during writing this article is **1.29.0**. After starting Minishift we need to run some additional oc commands to enable source-2-image for Java apps. First, we add some privileges to user admin to be able to access project openshift. In this project OpenShift stores all the build-in templates and image streams used, for example as S2I builders. Let's begin from enable admin-user addon.

```
$ minishift addons apply admin-user
```

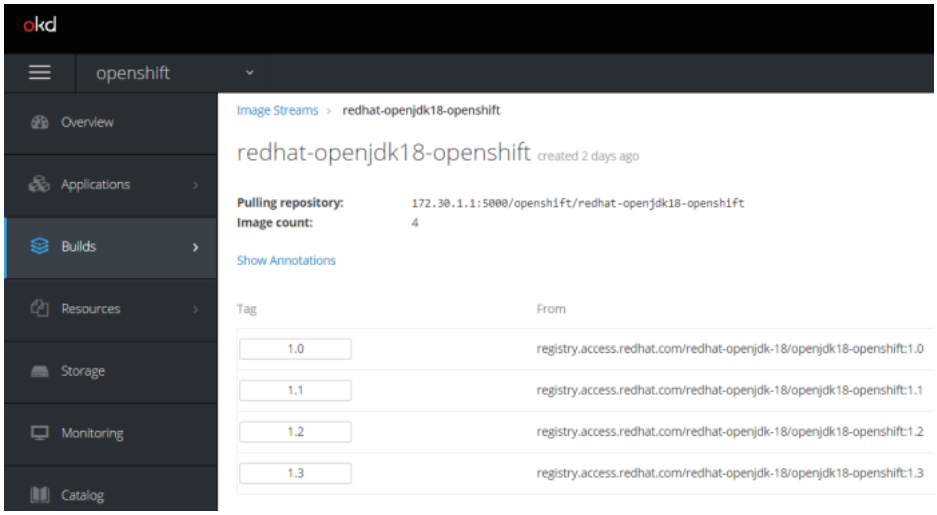
Thanks to that plugin we are able to login to Minishift as cluster admin. Now, we can grant role cluster-admin to user admin.

```
$ oc login -u system:admin
$ oc adm policy add-cluster-role-to-user cluster-admin admin
$ oc login -u admin -p admin
```

After that, you can login to the web console using credentials **admin/admin**. You should be able to see project openshift. It is not all. The image used for building runnable Java apps (openjdk18-openshift) is not available by default on Minishift. We can import it manually from RedHat registry using oc import-image command or just enable and apply plugin xpaas. I prefer the second option.

```
$ minishift addons apply xpaas
```

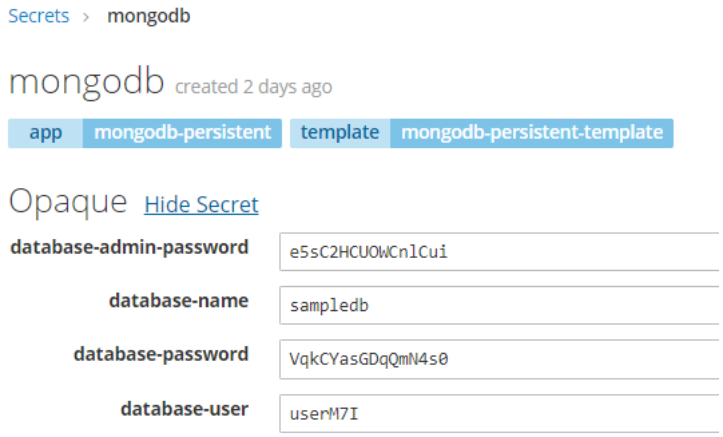
Now, you can go to Minishift web console (for me available under address https://192.168.99.100:8443), select project openshift and then navigate to *Builds -> Images*. You should see the image stream redhat-openjdk18-openshift on the list.



The newest version of that image is 1.3. Surprisingly it is not the newest version on OpenShift Container Platform. There you have version 1.5. However, the newest versions of builder images has been moved to registry.redhat.io, which requires authentication.

3. Deploying Java app using S2I

We are finally able to deploy our app on Minishift with S2I builder. The application source code is ready, and the same with the Minishift instance. The first step is to deploy an instance of MongoDB. It is very easy with OpenShift, because the Mongo template is available in the built-in service catalog. We can provide our own configuration settings or left default values. What's important for us, OpenShift generates secrets, by default available under the name mongodb.



The S2I builder image provided by OpenShift may be used through the image stream **redhat-openjdk18-openshift**. This image is intended for use with Maven-based Java standalone projects that are run via main class, for example Spring Boot applications. If you would not provide any builder during creating the new app the type of application is auto-detected by OpenShift, and source code written Java it will be deployed on WildFly server. The current version of the Java S2I builder image supports OpenJDK 1.8, Jolokia 1.3.5, and Maven 3.3.9-2.8.
Let's create our first application on OpenShift. We begin from microservice employee. Under normal circumstances each microservice would be located in a separate Git repository. In our sample all of them are placed in the single repository, so we have provided the location of the current app by setting parameter --context-dir. We will also override default branch to openshift, which has been created for the purposes of this article.

```
$ oc new-app redhat-openjdk18-openshift:1.3~https://github.com/piomin/sample-spring-microservices-new.git#openshift --name=employee --context-dir=employee-service
```

All our microservices are connected to the Mongo database, so we also have to inject connection settings and credentials into the application pod. It can be achieved by injecting mongodb secret to BuildConfig object.

```
$ oc set env bc/employee --from="secret/mongodb" --prefix=MONGO_
```

BuildConfig is one of the OpenShift object created after running command oc new-app. It also creates DeploymentConfig with deployment definition, Service, and ImageStream with newest Docker image of application. After creating the application a new build is running. First, it downloads source code from the Git repository, then it builds it using Maven, assembles, builds results into the Docker image, and finally saves the image in the registry.
Now, we can create the next application – department. For simplification, all three microservices are connecting to the same database, which is not recommended under normal circumstances. In that case the only difference between department and employee app is the environment variable EMPLOYEE_SERVICE set as parameter on oc new-app command.

```
$ oc new-app redhat-openjdk18-openshift:1.3~https://github.com/piomin/sample-spring-microservices-new.git#openshift --name=department --context-dir=department-service -e EMPLOYEE_SERVI
```

The same as before we also inject mongodb secret into BuildConfig object.

```
$ oc set env bc/department --from="secret/mongodb" --prefix=MONGO_
```

A build is starting just after creating a new application, but we can also start it manually by executing the following running command.

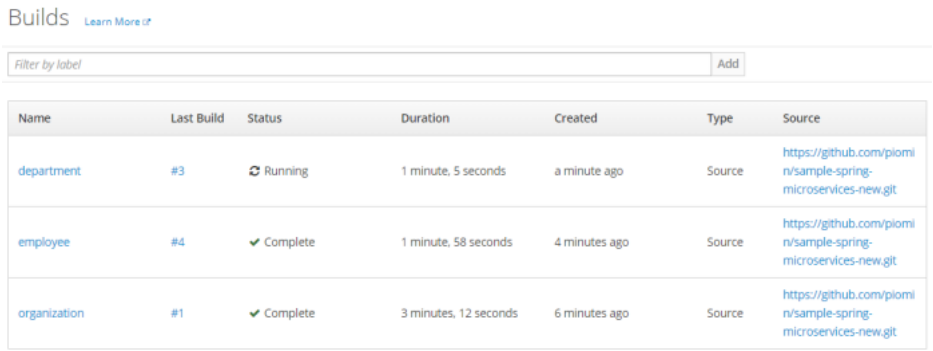
```
$ oc start-build department
```

Finally, we are deploying the last microservice. Here are the appropriate commands.

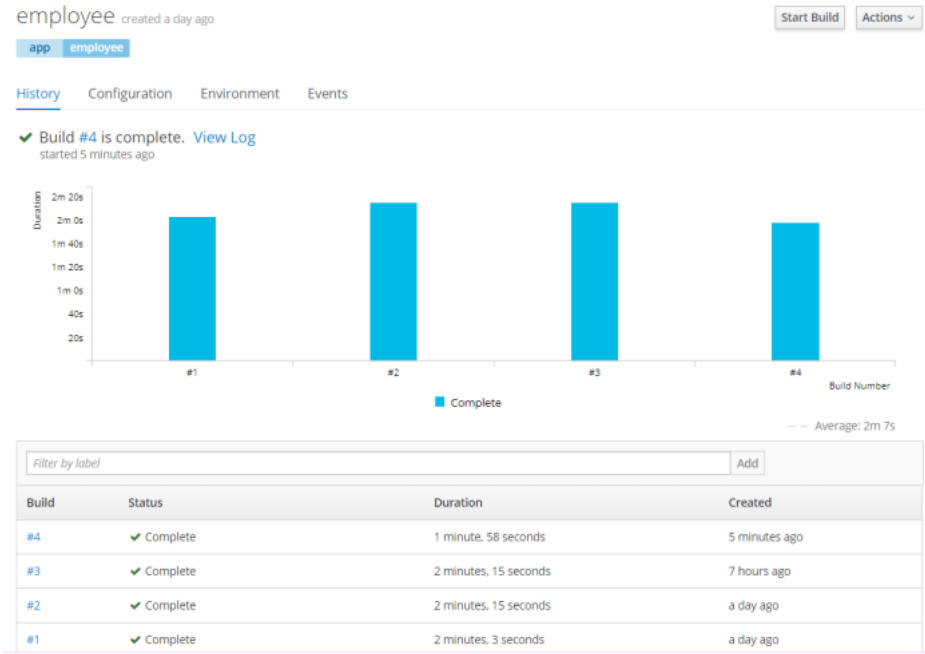
```
$ oc new-app redhat-openjdk18-openshift:1.3~https://github.com/piomin/sample-spring-microservices-new.git#openshift --name=organization --context-dir=organization-service -e EMPLOYEE_SER'
$ oc set env bc/organization --from="secret/mongodb" --prefix=MONGO_
```

4. Deep look into created OpenShift objects

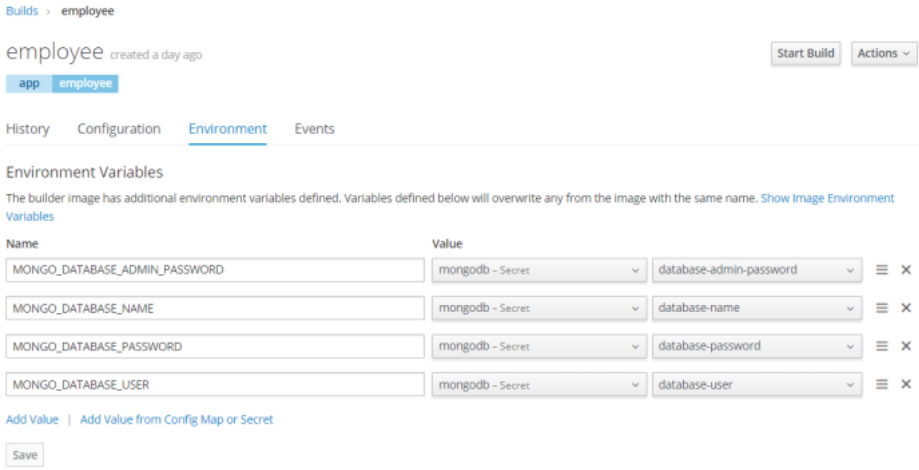
The list of builds may be displayed on the web console under section *Builds* -> *Builds*. As you can see on the picture below there are three BuildConfig objects available – each one for the single application. The same list can be displayed using oc command oc get bc.



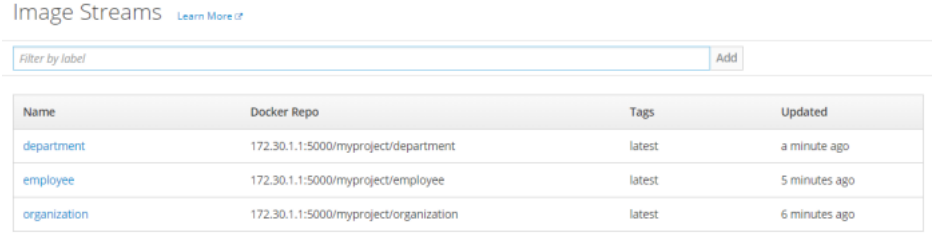
You can take a look at build history by selecting one of the elements from the list. You can also start a new by clicking the button *Start Build* as shown below.



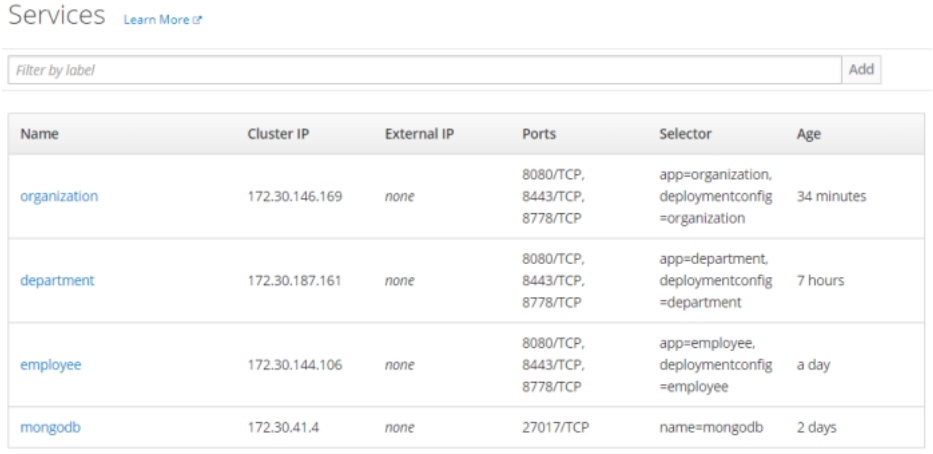
We can always display YAML configuration files with BuildConfig definition. But it is also possible to perform a similar action using a web console. The following picture shows the list of environment variables injected from mongodb secret into the BuildConfig object.



Every build generates a Docker image with an application and saves it in Minishift internal registry. Minishift internal registry is available under address *172.30.1.1:5000*. The list of available image streams is available under section *Builds -> Images*.



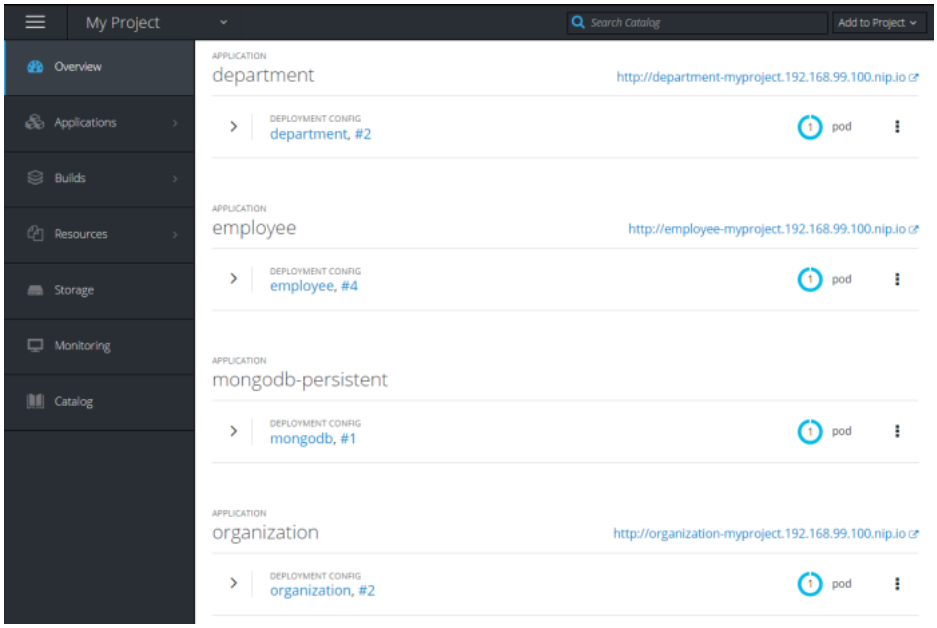
Every application is automatically exposed on ports 8080 (HTTP), 8443 (HTTPS) and 8778 (Jolokia) via services. You can also expose these services outside Minishift by creating OpenShift Route using oc expose command.



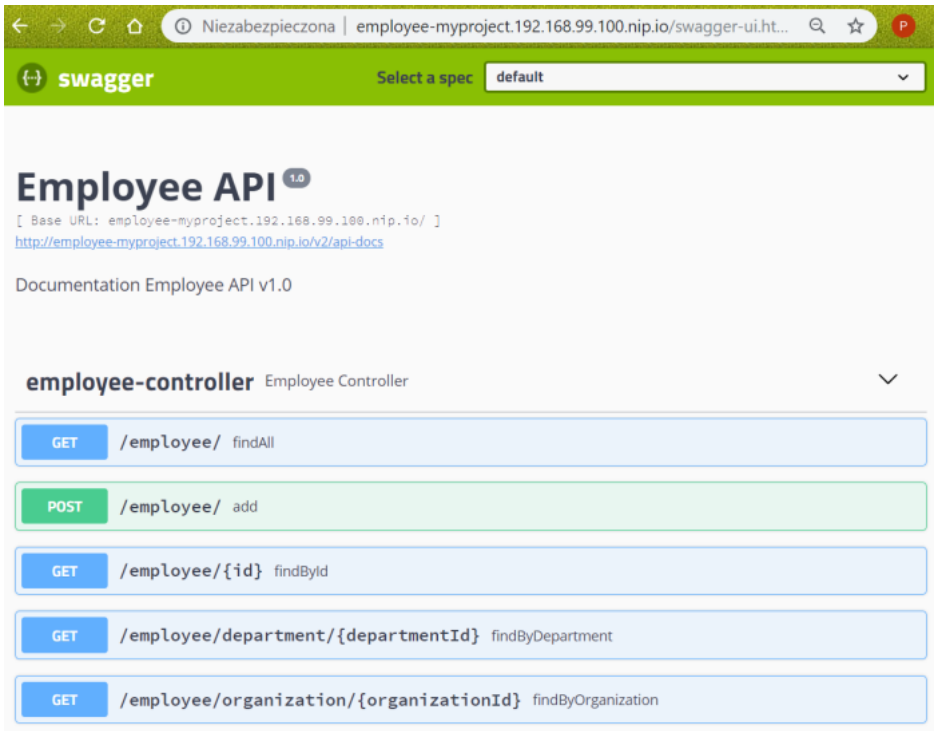
5. Testing the sample system

To proceed with the tests we should first expose our microservices outside Minishift. To do that just run the following commands.
$ oc expose svc employee
$ oc expose svc department
$ oc expose svc organization

After that we can access applications on the address http://${APP_NAME}-${PROJ_NAME}.${MINISHIFT_IP}.nip.io as shown below.



Each microservice provides Swagger2 API documentation available on page swagger-ui.html. Thanks to that we can easily test every single endpoint exposed by the service.



It's worth notice that every application making use of three approaches to inject environment variables into the pod:

1. It stores version number in source code repository inside the file .s2i/environment. S2I builder reads all the properties defined inside that file and set them as environment variables for builder pod, and then application pod. Our property name is VERSION, which is injected using Spring @Value, and set for Swagger API (the code is visible below).
2. I have already set the names of dependent services as ENV vars during executing command oc new-app for department and organization apps.
3. I have also inject MongoDB secret into every BuildConfig object using oc set env command.

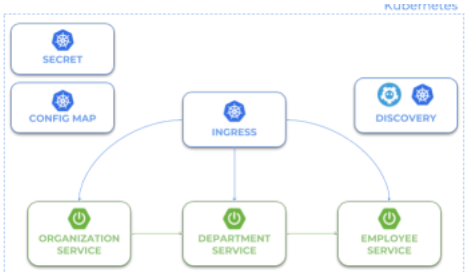
```
@Value("${VERSION}")
String version;

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

@Bean
public Docket swaggerApi() {
   return new Docket(DocumentationType.SWAGGER_2)
      .select()
        .apis(RequestHandlerSelectors.basePackage("pl.piomin.services.department.controller"))
        .paths(PathSelectors.any())
      .build()
      .apiInfo(new ApiInfoBuilder().version(version).title("Department API").description("Documentation Department API v" + version).build());
}
```

Conclusion

In this article I show you that deploying your applications on OpenShift may be a very simple thing. You don't have to create any YAML descriptor files or build Docker images by yourself to run your app. It is built directly from your source code. You can compare it with deployment on Kubernetes described in one of my previous articles Quick Guide to Microservices with Kubernetes, Spring Boot 2.0 and Docker.

Like this:
Like Loading...

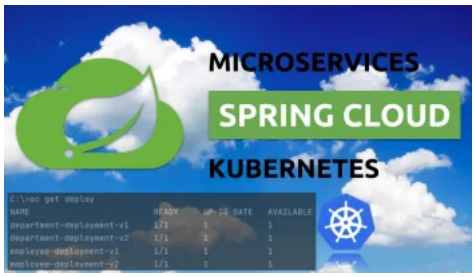
Related



Quick Guide to Microservices with Kubernetes, Spring Boot 2 and Docker

Here's the next article in a series of "Quick Guide to...". This time we will discuss and run examples of Spring Boot microservices on Kubernetes. The structure of that article will be quite similar to this one Quick Guide to Microservices with Spring Boot 2.0, Eureka and Spring Cloud, as…
August 2, 2018

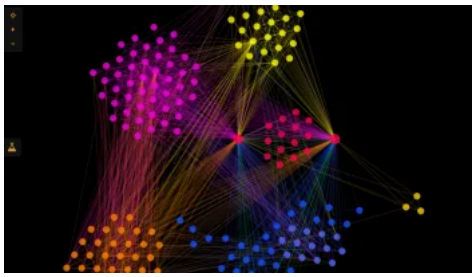
In "Containers"



Microservices With Spring Cloud Kubernetes

Spring Cloud and Kubernetes are the popular products applicable to various different use cases. However, when it comes to microservices architecture they are sometimes described as competitive solutions. They are both implementing popular patterns in microservices architecture like service discovery, distributed configuration, load balancing or circuit breaking. Of course, they…
December 20, 2019

In "Containers"



Microservices with Kubernetes and Docker

In one of my previous posts, I described an example of a continuous delivery configuration for building microservices with Docker and Jenkins. It was a simple configuration where I decided to use only Docker Pipeline Plugin for building and running containers with microservices. That solution had one big disadvantage -…
March 31, 2017

In "Containers"