Desenvolvimento de Aplicações com Arquitetura Baseada em Microservices

Prof. Vinicius Cardoso Garcia
vcg@cin.ufpe.br :: @vinicius3w :: assertlab.com

[IF1007] - Tópicos Avançados em SI 4
https://github.com/IF1007/if1007

AssertLab

Licença do material

Este Trabalho foi licenciado com uma Licença

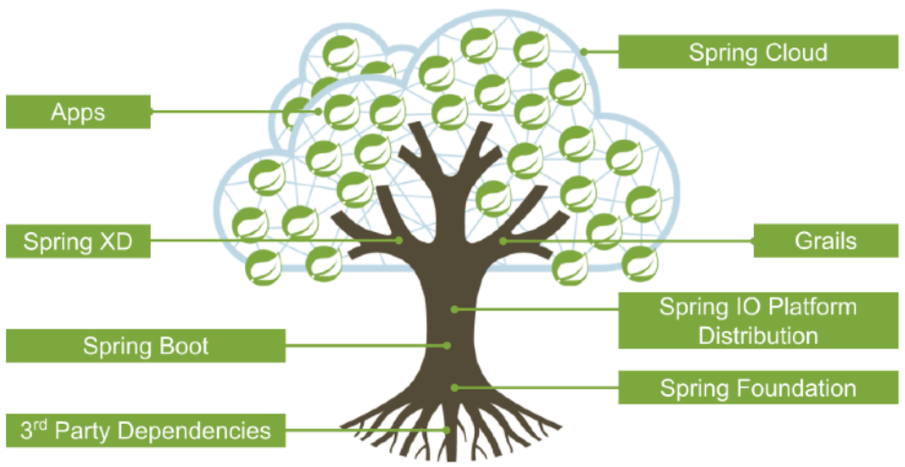
Creative Commons - Atribuição-NãoComercial-CompartilhaIgual 3.0 Não Adaptada

Mais informações visite

http://creativecommons.org/licenses/by-nc-sa/3.0/deed.pt

Resources

- There is no textbook required. However, the following are some books that may be recommended:
  - Building Microservices: Designing Fine-Grained Systems
  - Spring Microservices
  - Spring Boot: Acelere o desenvolvimento de microsserviços
  - Microservices for Java Developers A Hands-on Introduction to Frameworks and Containers
  - Migrating to Cloud-Native Application Architectures
  - Continuous Integration
  - Getting started guides from spring.io



Building Microservices with Spring Boot

Context

- Spring Boot is a framework to develop production-ready microservices in Java
- We will see..
  - Setting up the latest Spring development environment
  - Developing RESTful services using the Spring framework
  - Using Spring Boot to build fully qualified microservices
  - Useful Spring Boot features to build production-ready microservices

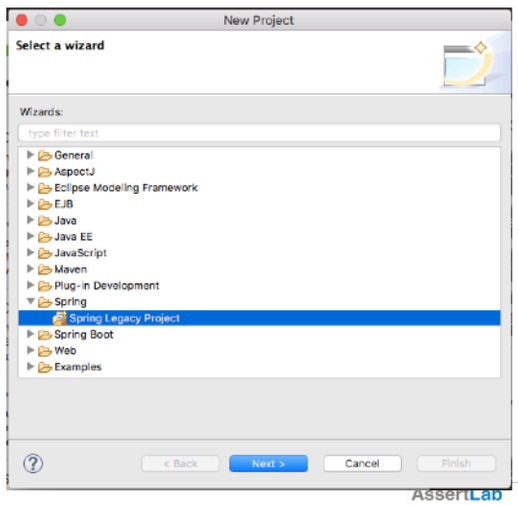
Setting up a development environment

- JDK 1.8: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
- Spring Tool Suite 3.9.2 (STS): https://spring.io/tools/sts/all
  - Alternately, other IDEs such as IntelliJ IDEA, NetBeans, or Eclipse could be used
- Maven 3.3.1: https://maven.apache.org/download.cgi
  - Similarly, alternate build tools such as Gradle can be used

- This class is based on the following versions of Spring libraries:
  - Spring Framework 4.2.6.RELEASE
  - Spring Boot 1.3.5.RELEASE

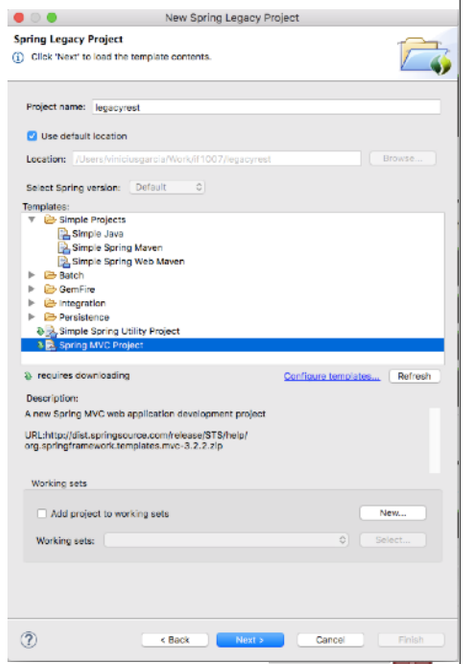
Developing a RESTful service – the legacy approach

- The following are the steps to develop the first RESTful service:
  - Start STS and set a workspace of choice for this project
  - Navigate to File | New | Project
  - Select Spring Legacy Project as shown in the following screenshot and click on Next:



Developing a RESTful service – the legacy approach

- Select Spring MVC Project as shown in the following diagram and click on Next:



Developing a RESTful service – the legacy approach

- Select a top-level package name of choice. This example uses `br.ufpe.cin.if1007.lec03.legacyrest` as the top-level package
- Then, click on Finish
- This will create a project in the STS workspace with the name legacyrest
- Before proceeding further, pom.xml needs editing

Developing a RESTful service – the legacy approach

- Change the Spring version to 4.2.6.RELEASE, as follows:

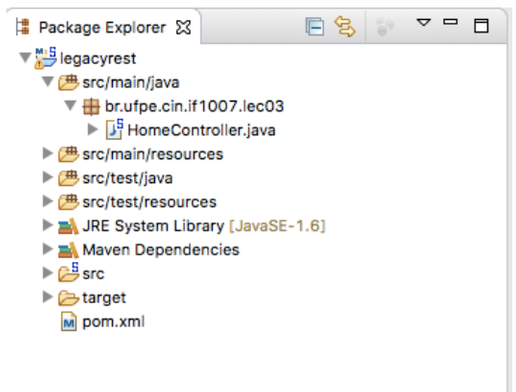
```
<org.springframework-version>4.2.6.RELEASE</org.springframework-version>
```

- Add Jackson dependencies in the pom.xml file for JSON-to-POJO and POJO-to-JSON conversions. Note that the 2.*.* version is used to ensure compatibility with Spring 4.

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.6.4</version>
</dependency>
```

Developing a RESTful service – the legacy approach

- Some Java code needs to be added. In Java Resources, under `legacyrest`, expand the package and open the default `HomeController.java` file:



Developing a RESTful service – the legacy approach

- To model the greeting representation, you create a resource representation class. Provide a plain old java object with fields, constructors, and accessors for the id and content data

legacyrest/pom.xml | HomeController.java | Greeting.java

```
package br.ufpe.cin;

public class Greeting {
    private String message;

    public Greeting(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

Developing a RESTful service – the legacy approach

- The default implementation is targeted more towards the MVC project. Rewriting `HomeController.java` to return a JSON value in response to the REST call will do the trick. The resulting `HomeController.java` file will look similar to the following:

```
package br.ufpe.cin.if1007.lec03;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

/**
 * Handles requests for the application home page.
 */
@RestController
public class HomeController {
    @RequestMapping("/")
    public Greet sayHello(){
        return new Greet("Bora BAÊA!");
    }
}

class Greet {
    private String message;
    public Greet(String message) {
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }

}
```

Developing a RESTful service – the legacy approach

- Examining the code, there are now two classes:
  - `Greeting`: This is a simple Java class with getters and setters to represent a data object. There is only one attribute in the `Greeting` class, which is `message`.
  - `HomeController.java`: This is nothing but a Spring controller REST endpoint to handle HTTP requests.

- Note that the annotation used in HomeController is `@RestController`, which automatically injects `@Controller` and `@ResponseBody` and has the same effect as the following code:

```
@Controller
@ResponseBody
public class HomeController { }
```

legacyrest/pom.xml | HomeController.java | Greeting.java

```
package br.ufpe.cin;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HomeController {

    @RequestMapping("/")
    public Greeting sayHello(){
        return new Greeting("Bora BAÉA!");
    }

}
```


The project can now be run by right-clicking on legacyrest, navigating to Run As | Run On Server, and then selecting the default server (Pivotal tc Server Developer Edition v3.1) that comes along with STS

Moving from traditional web applications to microservice

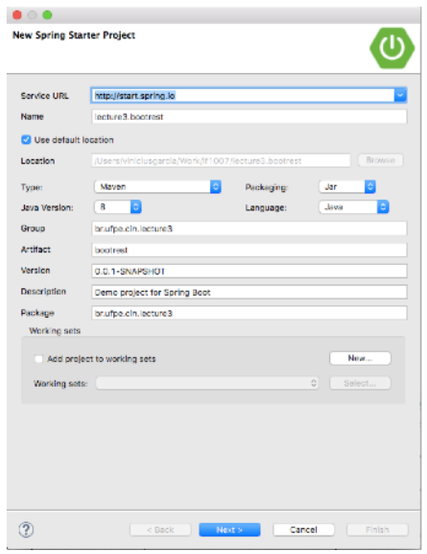
- At first glance, the preceding RESTful service is a fully qualified interoperable REST/JSON service.
- However, it is not fully autonomous in nature.
- This is primarily because the service relies on an underlying application server or web container.
- In the preceding example, a war was explicitly created and deployed on a Tomcat server

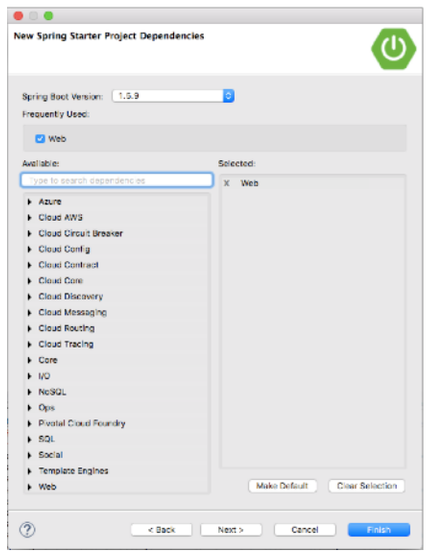
This is a traditional approach to developing RESTful services as a web application. However, from the microservices point of view, one needs a mechanism to develop services as executables, self-contained JAR files with an embedded HTTP listener.

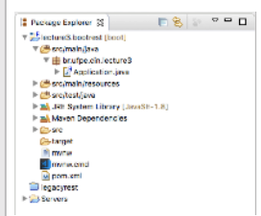
Using Spring Boot to build RESTful microservices

- The framework uses an opinionated approach over configurations for decision making, thereby reducing the effort required in writing a lot of boilerplate code and configurations
- Spring Boot only autoconfigures build files — for example, POM files in the case of Maven
- One of the great outcomes of Spring Boot is that it almost eliminates the need to have traditional XML configurations
- Enables microservices' development by packaging all the required runtime dependencies in a fat executable JAR file

Developing the Spring Boot Java microservice using STS







Examining the POM file

- The `spring-boot-starter-parent` pattern is a bill of materials (BOM), a pattern used by Maven's dependency management
- Reviewing the dependency section, one can see that this is a clean and neat POM file with only two dependencies
  - `spring-boot-starter-web` adds all dependencies required for a Spring MVC project
  - also includes dependencies to Tomcat as an embedded HTTP listener

lecture3.bootrest/pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>br.ufpe.cin.lecture3</groupId>
    <artifactId>bootrest</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>lecture3.bootrest</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.9.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

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

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

The advantage of using the spring-boot-starter-parent POM file is that developers need not worry about finding the right compatible versions of different libraries such as Spring, Jersey, JUnit, Logback, Hibernate, Jackson, and so on.

Examining Application.java

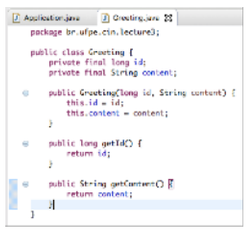
- There is only a main method in Application, which will be invoked at startup as per the Java convention
  - calling the run method on SpringApplication. Application.class
- The magic is done by the @SpringBootApplication annotation
  - @Configuration
  - @EnableAutoConfiguration
  - @ComponentScan

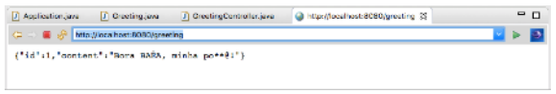
lecture3.bootrest/pom.xml | Application.java

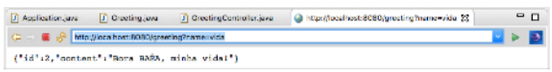
```
package br.ufpe.cin.lecture3;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

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

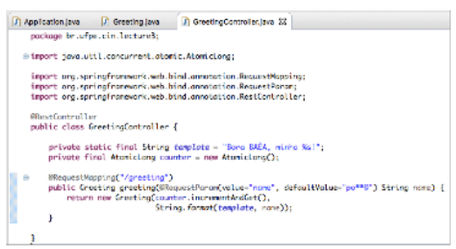

The @Configuration annotation hints that the contained class declares one or more @Bean definitions. The @Configuration annotation is meta-annotated with @Component; therefore, it is a candidate for component scanning.
The @EnableAutoConfiguration annotation tells Spring Boot to automatically configure the Spring application based on the dependencies available in the class path.

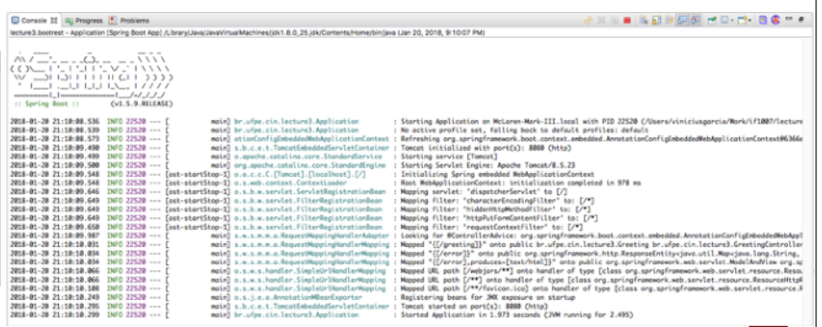
Developing the Spring Boot Java microservice using STS











Testing the Spring Boot microservice

- There are multiple ways to test REST/JSON Spring Boot microservices
  - The easiest way is to use a web browser or a curl command pointing to the URL

  - There are number of tools available to test RESTful services, such as Postman, Advanced REST client, SOAP UI, Paw, and so on

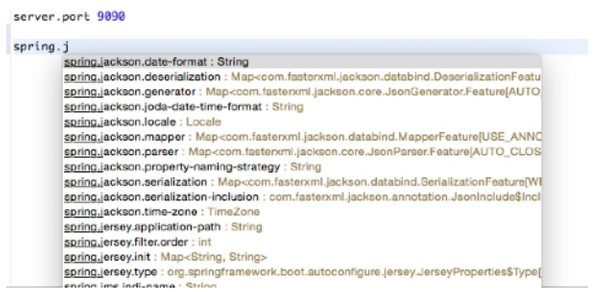
The Spring Boot configuration

- Understanding the Spring Boot autoconfiguration
  - Convention over configuration by scanning the dependent libraries available in the class path
  - For each spring-boot-starter-* dependency in the POM file, Spring Boot executes a default AutoConfiguration class
  - It is possible to exclude the autoconfiguration of certain libraries

```
@EnableAutoConfiguration(exclude=
{DataSourceAutoConfiguration.class})
```

The Spring Boot configuration

- Overriding default configuration values
  - It is also possible to override default configuration values using the application.properties file
    - STS provides an easy-to-autocomplete, contextual help on application.properties
  - server.port is edited to be set as 9090. Running this application again will start the server on port 9090.



The Spring Boot configuration

- Changing the location of the configuration file
  - In order to align with the Twelve-Factor app, configuration parameters need to be externalized from the code
    - Spring Boot externalizes all configurations into application.properties. However, it is still part of the application's build. Furthermore, properties can be read from outside the package ~> spring.config.location could be a local file location

```
spring.config.name= # config file name
spring.config.location= # location of config file
```

- The following command starts the Spring Boot application with an externally provided configuration file:

```
$java -jar target/bootadvanced-0.0.1-SNAPSHOT.jar --
spring.config.name=bootrest.properties
```

The Spring Boot configuration

- Reading custom properties
  - At startup, SpringApplication loads all the properties and adds them to the Spring Environment class
    - Autowire the Spring Environment class into the GreetingController class.
    - Edit the GreetingController class to read the custom property from Environment and add a log statement to print the custom property to the console

1. Add the following property to the `application.properties` file:

```
bootrest.customproperty=hello
```

2. Then, edit the `GreetingController` class as follows:

```
@Autowired
Environment env;

Greet greet(){
    logger.info("bootrest.customproperty
"+ env.getProperty("bootrest.customprop-
erty"));
    return new Greet("Hello World!");
}
```

3. Rerun the application. The log statement prints the custom variable in the console, as follows:

```
org.rvslab.chapter2.GreetingController
: bootrest.customproperty hello
```

The Spring Boot configuration

- Using a .yaml file for configuration
  - simply replace application.properties with application.yaml and add the following property

```
server
  port: 9080
```

The Spring Boot configuration

- Using multiple configuration profiles
  - It is possible to have different profiles such as development, testing, staging, production, and so on

```
"mvn -Dspring.profiles.active=production install
mvn -Dspring.profiles.active=development install
```

  - Active profiles can be specified programmatically using the @ActiveProfiles annotation ~> test cases

```
@ActiveProfiles("test")
```

```
spring:
    profiles: development
server:
      port: 9090
---

spring:
    profiles: production
server:
      port: 8080
```

Changing the default embedded web server

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-undertow</artifactId>
</dependency>
```


Embedded HTTP listeners can easily be customized as follows. By default, Spring Boot supports Tomcat, Jetty, and Undertow. In the following example, Tomcat is replaced with Undertow

Implementing Spring Boot security

- Securing microservices with basic security

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

```
@EnableGlobalMethodSecurity
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```


Open Application.java and add @EnableGlobalMethodSecurity to the Application class. This annotation will enable method-level security

Implementing Spring Boot security

- Securing microservices with basic security
  - The default basic authentication assumes the user as being user. The default password will be printed in the console at startup.
  - Alternately, the username and password can be added in application.properties

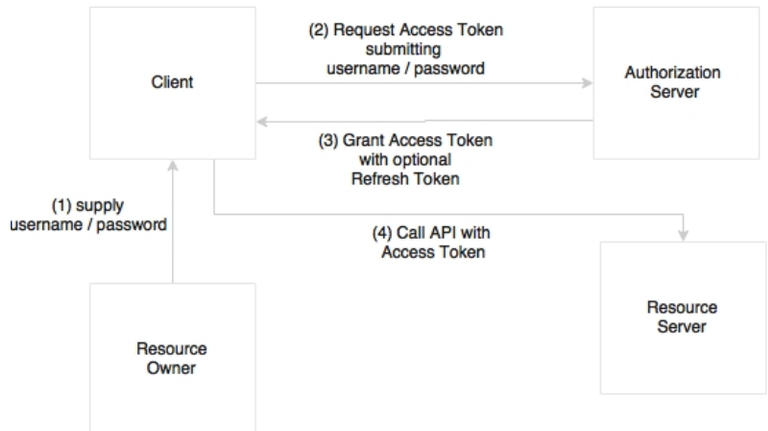
```
security.user.name=guest
security.user.password=guest123
```

```
  @Test
  public void testSecureService() {
    String plainCreds = "guest:guest123";
    HttpHeaders headers = new HttpHeaders();
    headers.add("Authorization", "Basic " +
new String(Base64.encode(plainCreds.get-
Bytes())));
    HttpEntity<String> request = new HttpEn-
tity<String>(headers);
    RestTemplate restTemplate = new RestTem-
plate();

    ResponseEntity<Greet> response = rest-
Template.exchange("http://localhost:8080",
HttpMethod.GET, request, Greet.class);
    Assert.assertEquals("Hello World!",
response.getBody().getMessage());
  }
```


As shown in the code, a new Authorization request header with Base64 encoding the username-password string is created.
Rerun the application using Maven. Note that the new test case passed, but the old test case failed with an exception. The earlier test case now runs without credentials, and as a result, the server rejected the request with the following message:
org.springframework.web.client.HttpClientErrorException: 401 Unauthorized

Implementing Spring Boot security

- Securing a microservice with OAuth2



The resource owner provides the client with a username and password. The client then sends a token request to the authorization server by providing the credential information. The authorization server authorizes the client and returns with an access token. On every subsequent request, the server validates the client token

Securing a microservice with OAuth2

- As a first step, update pom.xml with the OAuth2 dependency, as follows:

```
<dependency>
<groupId>org.springframework.security.
oauth</groupId>

  <artifactId>spring-security-oauth2</
artifactId>

  <version>2.0.9.RELEASE</version>

</dependency>
```

- Next, add two new annotations, @EnableAuthorizationServer and @EnableResourceServer, to the Application.java file

```
@EnableResourceServer

@EnableAuthorizationServer

@SpringBootApplication

public class Application {
```

- Add the following properties to the application.properties file:

```
security.user.name=guest

security.user.password=guest123

security.oauth2.client.clientId:
trustedclient

security.oauth2.client.clientSecre
t: trustedclient123

security.oauth2.client.authorized-
grant-types:
authorization_code,refresh_token,p
assword

security.oauth2.client.scope:
openid
```


The **@EnableAuthorizationServer** annotation creates an authorization server with an in-memory repository to store client tokens and provide clients with a username, password, client ID, and secret.
The **@EnableResourceServer** annotation is used to access the tokens. This enables a spring security filter that is authenticated via an incoming OAuth2 token.
In our example, both the authorization server and resource server are the same.

- Then, add another test case to test OAuth2

```
@Test
  public void testOAuthService() {
"ResourceOwnerPasswordResourceDetails resource = new ResourceOwnerPasswordResourceDetails();
        resource.setUsername("guest");
        resource.setPassword("guest123");
          resource.setAccessTokenUri("http://localhost:8080/oauth/token");
        resource.setClientId("trustedclient");
        resource.setClientSecret("trustedclient123");
        resource.setGrantType("password");

        DefaultOAuth2ClientContext clientContext = new DefaultOAuth2ClientContext();
        OAuth2RestTemplate restTemplate = new OAuth2RestTemplate(resource, clientContext);

        Greet greet = restTemplate.getForObject("http://localhost:8080", Greet.class);

        Assert.assertEquals("Hello World!", greet.getMessage());
  }
```


As shown in the preceding code, a special REST template, OAuth2RestTemplate, is created by passing the resource details encapsulated in a resource details object. This REST template handles the OAuth2 processes underneath. The access token URI is the endpoint for the token access.

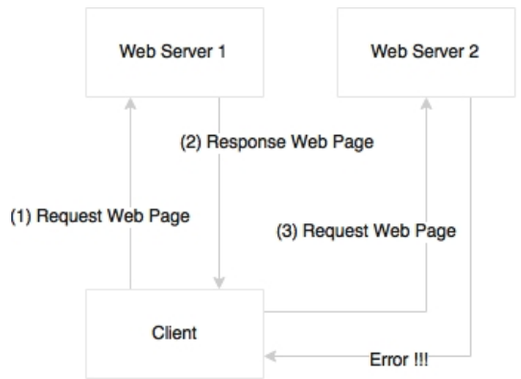
Securing a microservice with OAuth2

- Rerun the application using mvn install.
  - The first two test cases will fail, and the new one will succeed.
  - This is because the server only accepts OAuth2-enabled requests.

- These are quick configurations provided by Spring Boot out of the box but are not good enough to be production grade.
  - We may need to customize ResourceServerConfigurer and AuthorizationServerConfigurer to make them production-ready.

Enabling cross-origin access for microservices

- Browsers are generally restricted when client-side web applications running from one origin request data from another origin
- Enabling cross-origin access is generally termed as CORS (Cross-Origin Resource Sharing)
- With microservices, as each service runs with its own origin, it will easily get into the issue of a client-side web application consuming data from multiple origins



Enabling cross-origin access for microservices

- Spring Boot provides a simple declarative approach to enabling cross-origin requests
  - By default, all the origins and headers are accepted

```
@RestController
class GreetingController{
  @CrossOrigin
  @RequestMapping("/")
  Greet greet(){
    return new Greet("Hello World!");
  }
}
```

- The @CrossOrigin annotation enables a method or class to accept cross-origin requests

```
@CrossOrigin("http://mytrustedorigin.com")
```

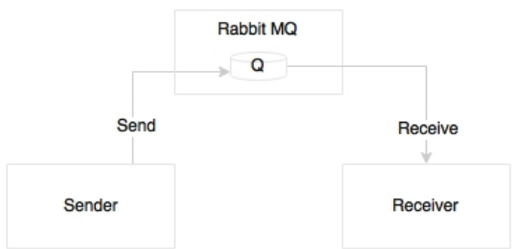
- Global CORS can be enabled using the WebMvcConfigurer bean and customizing the addCorsMappings(CorsRegistry registry) method

“For instance, a scenario where a browser client accessing Customer from the Customer microservice and Order History from the Order microservices is very ”

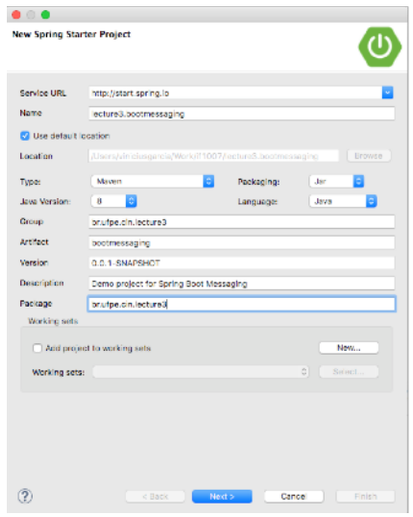
Excerpt From: “Spring Microservices.” iBooks.

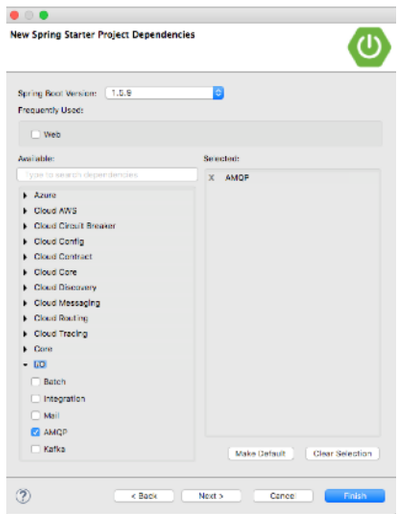
Implementing Spring Boot messaging

- In an ideal case, all microservice interactions are expected to happen asynchronously using publish-subscribe semantics.
  - Spring Boot provides a hassle-free mechanism to configure messaging solutions
- In this example, we will create a Spring Boot application with a sender and receiver, both connected though an external queue



Implementing Spring Boot messaging





Rabbit MQ will also be needed for this example. Download and install the latest version of Rabbit MQ from https://www.rabbitmq.com/download.html

Implementing Spring Boot messaging

- Follow the installation steps documented on the site. Once ready, start the RabbitMQ server via the following command:
  - `$./rabbitmq-server`
- Make the configuration changes to the application.properties file to reflect the RabbitMQ configuration. The following configuration uses the default port, username, and password of RabbitMQ:
  - `spring.rabbitmq.host=localhost`
  - `spring.rabbitmq.port=5672`
  - `spring.rabbitmq.username=guest`
  - `spring.rabbitmq.password=guest`

Implementing Spring Boot messaging

- Add a message sender component and a queue named `TestQ` of the `org.springframework.amqp.core.Queue` type to the `Application.java` file under `src/main/java`. `RabbitMessagingTemplate` is a convenient way to send messages, which will abstract all the messaging semantics

Dashboard | application.properties | Sender.java

```
package br.ufpe.cin.lecture3;

import org.springframework.amqp.core.Queue;
import org.springframework.amqp.rabbit.core.RabbitMessagingTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.stereotype.Component;

@Component
public class Sender {

    RabbitMessagingTemplate template;

    @Autowired
    Sender(RabbitMessagingTemplate template){
        this.template = template;
    }
    @Bean
    Queue queue() {
        return new Queue("TestQ", false);
    }

    public void send(String message){
        template.convertAndSend("TestQ", message);
    }

}
```

AssertLab

Spring Boot provides all boilerplate configurations to send messages

Implementing Spring Boot messaging

- To receive the message, all that needs to be used is a `@RabbitListener` annotation. Spring Boot autoconfigures all the required boilerplate configurations

```java
package br.ufpe.cin.lecture3;

import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

@Component
public class Receiver {

    @RabbitListener(queues = "TestQ")
    public void processMessage(String content) {
       System.out.println(content);
    }

}
```

Implementing Spring Boot messaging

- The last piece of this exercise is to wire the sender to our main application and implement the `run` method of `CommandLineRunner` to initiate the message sending. When the application is initialized, it invokes the `run` method of `CommandLineRunner`
- Run the application as a Spring Boot application and verify the output. The following message will be printed in the console

Sender.java | Receiver.java | Application.jav

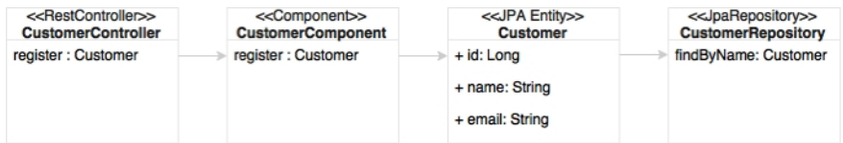
```
package br.ufpe.cin.lecture3;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.CommandLineRunner;


@SpringBootApplication
public class Application implements CommandLineRunner{

    @Autowired
    Sender sender;

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

    @Override
    public void run(String... args) throws Exception {
    sender.send("Bora BAÊA..!!!");
    }
}
```


Essentially, what's being done is this:
.put((a, b, c) => controller.update(a, b, c))
Of course, what if we want 4 parameters, or 5, or 6? We don't want to write a new version of the function for all possible quantities of parameters.
The spread operator (...) allows us to accept a variable number of arguments and store them in an array.

Developing a comprehensive microservice example

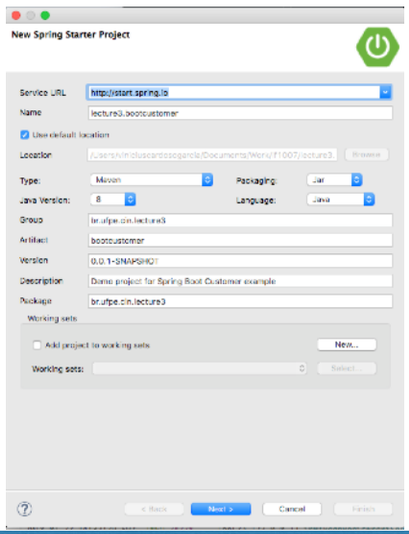
- The Customer Profile microservice exposes methods to **create**, **read**, **update**, and **delete** (**CRUD**) a customer and a registration service to register a customer.
- The registration process applies certain business logic, saves the customer profile, and sends a message to the Customer Notification microservice.
- The Customer Notification microservice accepts the message sent by the registration service and sends an e-mail message to the customer using an SMTP server.
- Asynchronous messaging is used to integrate Customer Profile with the Customer Notification service

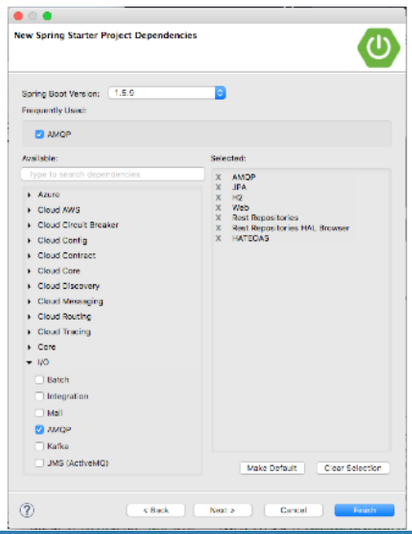
Developing a comprehensive microservice example

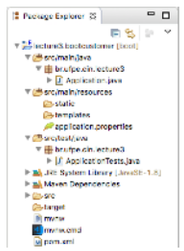


- `CustomerController` in the diagram is the REST endpoint, which invokes a component class, `CustomerComponent`. The component class/bean handles all the business logic. `CustomerRepository` is a Spring data JPA repository defined to handle the persistence of the `Customer` entity

Developing a comprehensive microservice example







Developing a comprehensive microservice example

- Start building the application by adding an Entity class named `Customer`. For simplicity, there are only three fields added to the `Customer` Entity class: the autogenerated `id` field, `name`, and `email`

Developing a comprehensive microservice example

- Add a repository class to handle the persistence handling of Customer. `CustomerRepository` extends the standard JPA repository. This means that all CRUD methods and default finder methods are automatically implemented by the Spring Data JPA repository

Dashboard | Customer.java | CustomerRepository.java

```
package br.ufpe.cin.lecture3;

import java.util.Optional;

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

@RepositoryRestResource
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    Optional<Customer> findByName(@Param("name") String name);

}
```

See more information about HATEOAS at https://spring.io/understanding/HATEOAS

In this example, we added a new method to the repository class, findByName, which essentially searches the customer based on the customer name and returns a Customer object if there is a matching name

The @RepositoryRestResource annotation enables the repository access through RESTful services. This will also enable HATEOAS and HAL by default. As for CRUD methods there is no additional business logic required, we will leave it as it is without controller or component classes. Using HATEOAS will help us navigate through Customer Repository methods effortlessly.
Note that there is no configuration added anywhere to point to any database. As H2 libraries are in the class path, all the configuration is done by default by Spring Boot based on the H2 autoconfiguration.

Developing a comprehensive microservice example

- Update the `Application.java` file by adding `CommandLineRunner` to initialize the repository with some customer records

Dashboard | Customer.java | CustomerRepository.java | Application.java

```
package br.ufpe.cin.lecture3;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.boot.CommandLineRunner;

@SpringBootApplication
public class Application {

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

    @Bean
    CommandLineRunner init(CustomerRepository customerRepository) {
        return (evt) -> {
            customerRepository.save(new Customer("Raudinei","raudinei@baea.com"));
            customerRepository.save(new Customer("Bobo","bobo@baea.com"));
            customerRepository.save(new Customer("Charles","charles@baea.com"));
            customerRepository.save(new Customer("Tarantini","tarantini@baea.com"));
            customerRepository.save(new Customer("Nonato","nonato@baea.com"));
            customerRepository.save(new Customer("Paulo Rodrigues","prodrigues@baea.com"));
            customerRepository.save(new Customer("Zé Carlos","zecarlos@baea.com"));
        };
    }
}
```

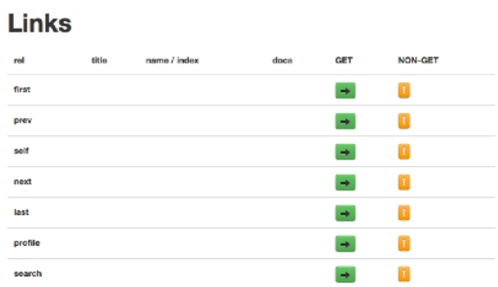
AssertLab

Developing a comprehensive microservice example

- Run the application as Spring Boot App. Open the HAL browser and point the browser to http://localhost:8080
- In the **Explorer** section, point to http://localhost:8080/customers and click on **Go**. This will list all the customers in the **Response Body** section of the HAL browser
- In the **Explorer** section, enter http://localhost:8080/customers?size=2&page=1&sort=name and click on **Go**. This will automatically execute paging and sorting on the repository and return the result
  - As the page size is set to `2` and the first page is requested, it will come back with two records in a sorted order

Developing a comprehensive microservice example

- Review the Links section. As shown in the following screenshot, it will facilitate navigating **first**, **next**, **prev**, and **last**. These are done using the HATEOAS links automatically generated by the repository browser



Developing a comprehensive microservice example

- Add a controller class, `CustomerController`, to handle service endpoints. There is only one endpoint in this class, `/register`, which is used to register a customer. If successful, it returns the `Customer` object as the response

```
package br.ufpe.cin.lecture3;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class CustomerController {

CustomerRegistrar customerRegistrar;

    @Autowired
    CustomerController(CustomerRegistrar customerRegistrar){
        this.customerRegistrar = customerRegistrar;
    }

    @RequestMapping( path="/register", method = RequestMethod.POST)
    Customer register(@RequestBody Customer customer){
        return customerRegistrar.register(customer);
    }

}
```

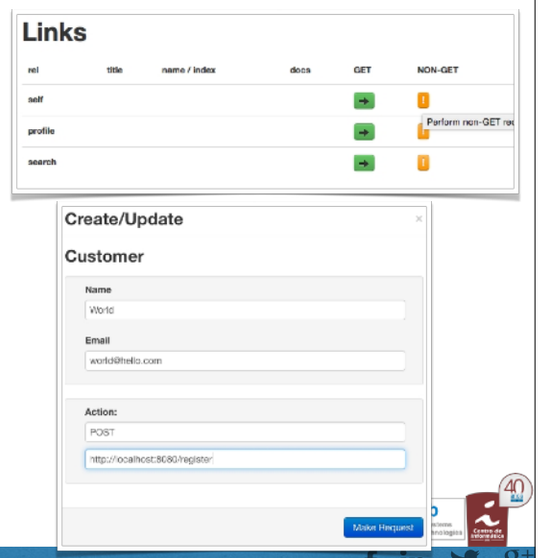
Developing a comprehensive microservice example

- A `CustomerRegistrar` component is added to handle the business logic.
- In this component class, while registering a customer, we will just check whether the customer name **already exists** in the database or not.
- If it does not exist, then we will insert a new record, and otherwise, we will send an error message back

```java
package br.ufpe.cin.lecture3;

import java.util.Optional;

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

@Component
public class CustomerRegistrar {

    CustomerRepository customerRespository;

    @Autowired
    CustomerRegistrar(CustomerRepository customerRespository){
        this.customerRespository = customerRespository;;
    }

    Customer register(Customer customer){
        Optional<Customer> existingCustomer = customerRespository.findByName(customer.getName());
        if (existingCustomer.isPresent()){
            throw new RuntimeException("is already exists");
        } else {
            customerRespository.save(customer);
        }
        return customer;
    }

}
```

Developing a comprehensive microservice example

- Restart the Boot application and test using the HAL browser via the URL http://localhost:8080
- Point the **Explorer** field to http://localhost:8080/customers
  - Review the results in the **Links** section
- Click on the **NON-GET** option against self. This will open a **form** to create a **new customer**



Fill the form and change the Action as shown in the diagram. Click on the Make Request button. This will call the register service and register the customer. Try giving a duplicate name to test the negative case

Developing a comprehensive microservice example

- Now, integrating the Customer Notification service to notify the customer
- Update `CustomerRegistrar` to call the second service. This is done through messaging. In this case, we injected a `Sender` component to send a notification to the customer by passing the customer's e-mail address to the sender

```java
package br.ufpe.cin.lecture3;

import java.util.Optional;

@Component
public class CustomerRegistrar {

    CustomerRepository customerRespository;
    Sender sender;

    @Autowired
    CustomerRegistrar(CustomerRepository customerRespository, Sender sender){
        this.customerRespository = customerRespository;
        this.sender = sender;
    }

    Customer register(Customer customer){
        Optional<Customer> existingCustomer = customerRespository.findByName(customer.getName());
        if (existingCustomer.isPresent()){
            throw new RuntimeException("is already exists");
        } else {
            customerRespository.save(customer);
            sender.send(customer.getEmail());
        }
        return customer;
    }

}
```

Developing a comprehensive microservice example

- The sender component will be based on RabbitMQ and AMQP. In this example, `RabbitMessagingTemplate` is used as explored in the last messaging example
- The `@Lazy` annotation is a useful one and it helps to increase the boot startup time.
- These beans will be initialized only when the need arises

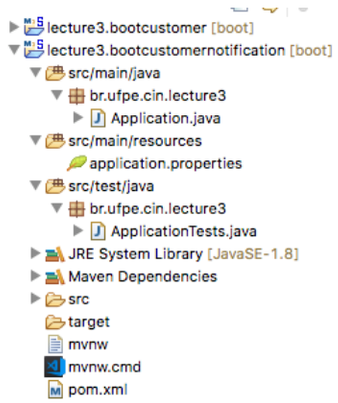
```
package br.ufpe.cin.lecture3;

import org.springframework.amqp.core.Queue;
import org.springframework.amqp.rabbit.core.RabbitMessagingTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.stereotype.Component;
import org.springframework.context.annotation.Lazy;

@Component
@Lazy
public class Sender {
    RabbitMessagingTemplate template;

    @Autowired
    Sender(RabbitMessagingTemplate template){
        this.template = template;
    }

    @Bean
    Queue queue() {
        return new Queue("CustomerQ", false);
    }

    public void send(String message){
        template.convertAndSend("CustomerQ", message);
    }

}
```

Developing a comprehensive microservice example

- We will also update the application.property file to include Rabbit MQ-related properties
  - `spring.rabbitmq.host=localhost`
  - `spring.rabbitmq.port=5672`
  - `spring.rabbitmq.username=guest`
  - `spring.rabbitmq.password=guest`

Developing a comprehensive microservice example

- To consume the message and send e-mails, we will create a notification service.
- For this, let's create another Spring Boot service, `lecture3.bootcustomernotification`.
  - Make sure that the **AMQP** and **Mail** starter libraries are selected when creating the Spring Boot service. Both **AMQP** and **Mail** are under **I/O**.



Developing a comprehensive microservice example

- Add a `Receiver` class. The `Receiver` class waits for a message on customer.
- This will receive a message sent by the Customer Profile service. On the arrival of a message, it sends an e-mail

```
package br.ufpe.cin.lecture3;

import org.springframework.amqp.core.Queue;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.stereotype.Component;

@Component
public class Receiver {

    @Autowired
    Mailer mailer;

    @Bean
    Queue queue(){
        return new Queue("CustomerQ", false);
    }

    @RabbitListener(queues = "CustomerQ")
    public void processMessage(String email) {
        System.out.println(email);
        mailer.sendMail(email);
    }

}
```

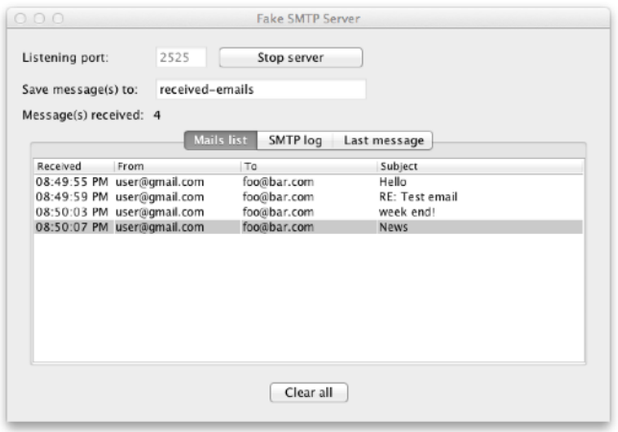
Developing a comprehensive microservice example

- Add another component to send an e-mail to the customer. We will use `JavaMailSender` to send an e-mail via code
- Behind the scenes, Spring Boot automatically configures all the parameters required by `JavaMailSender`

```
package br.ufpe.cin.lecture3;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.mail.SimpleMailMessage;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.stereotype.Component;

@Component
public class Mailer {

    @Autowired
    private  JavaMailSender  javaMailService;

    public void sendMail(String email){
      SimpleMailMessage mailMessage=new SimpleMailMessage();
      mailMessage.setTo(email);
      mailMessage.setSubject("Registration");
      mailMessage.setText("Successfully Registered");
      javaMailService.send(mailMessage);
    }

}
```

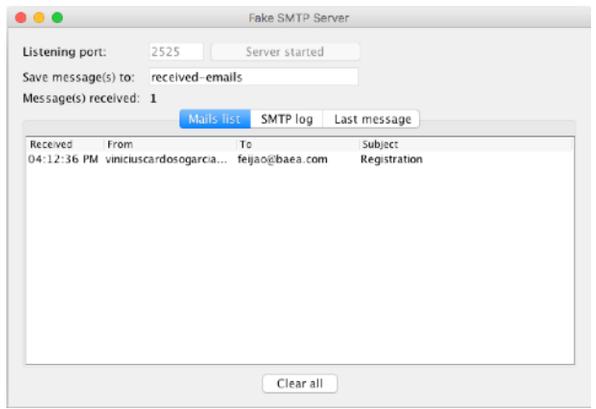
Developing a comprehensive microservice example

- To test SMTP, a test setup for SMTP is required to ensure that the mails are going out. In this example, FakeSMTP will be used. You can download FakeSMTP from http://nilhcem.github.io/FakeSMTP
- Once you download `fakeSMTP-2.0.jar`, run the SMTP server



Developing a comprehensive microservice example

- Start both the Spring Boot apps. Open the browser and repeat the customer creation steps through the HAL browser. In this case, immediately **after** submitting the request, we will be able to see the e-mail in the SMTP GUI.
- Internally, the Customer Profile service **asynchronously** calls the Customer Notification service, which, in turn, sends the e-mail message to the SMTP server



Spring Boot actuators

- We explored most of the Spring Boot features required to develop a microservice. Now, some of the production-ready operational aspects of Spring Boot need to be be explored

- Spring Boot actuators provide an excellent **out-of-the-box** mechanism to monitor and manage Spring Boot applications in production

Homework 3.1

- Browse the spring.io/guides, follow to the **Building a RESTful Web Service with Spring Boot Actuator**
- Show your results and impressions

Homework 3.2

- Documenting microservices
  - The traditional approach of API documentation is either by writing service specification documents or using static service registries. With a large number of microservices, it would be hard to keep the documentation of APIs in sync.
  - Microservices can be documented in many ways. This homework intend to explore how microservices can be documented using the popular **Swagger** framework.
  - Create a new **Spring Starter Project** and select **Web** in the library selection window. Name the project `lecture3.swagger`. Learn how to use **Springfox Swagger** library e using a similar `bootrest` project, show the results.

Documenting microservices with Springfox Swagger