# Prerequisites

1. You need to have a solid understanding of OO concepts and at least some programming background in Java language. Also you have to be familiar with relational database management concepts and be able to write at least basic DDL and DML commands as well as queries.
2. You have to install and IDE and familiarize yourself with it. Most members of the team use IntelliJ IDEA (free community edition is good enough) or Eclipse.
3. You need to have at least basic understanding on the tools/technics/concepts below:  
   Spring, Spring boot, Apache Kafka, HTTP and REST, JPA, Git, unit testing, micro services architecture

# The simplest Springboot app

Create a very simple Springboot app. Let’s have the groupId as “com.oracle.oal.microservices”, the artifactId/name as “onboarding” , the Java version 8 and the build/management tool is maven.

The only dependency needed for now is spring-boot-starter-web

Open the app in the IDE of your choice and familiarize yourself with the package structures and generated files

Tip: while it can be created manually, the [start.spring.io](file:///C:\Users\tkoszcze\AppData\Roaming\Microsoft\Word\start.spring.io) makes it super easy and quick

# Our first webb application

Turn your new app into a simple web application. The goal is that when a GET request is sent to the localhost:8080 URL, the following text is the response: Welcome to Oracle Application Labs!

Note: let’s have the response as plain text for now. In our real applications the response is usually a Json (generated automatically by Spring). Generating html pages is never needed.

Tip: there is probably more than one way to achieve this. Below you can see the most simple and elegant way:

package com.oracle.oal.microservices.onboarding;

import org.springframework.web.bind.annotation.GetMapping;

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

@RestController

public class FirstController {

@GetMapping("/")

public String greeting() {

return "Welcome to Oracle Application Labs!";

}

}

As you can see, with just 2 simple annotation put on a POJO, Srping MVC is able to map the incoming HTTP request to the appropriate handler, and also wrap the return value of a method into an HTTP request.

# HTTP request with parameters

Try out how to handle request parameters. For example let’s make the method extract the name of the user to do a personal greeting. You can do simply this:

public String greeting(@RequestParam String name) {

return name + ", welcome to Oracle Application Labs!";

}

Or a more tricky one:

public String greeting(@RequestParam(name = "name", required = false) Optional<String> optionalName) {

return optionalName.orElse("Mysterious newcomer") + ", welcome to Oracle Application Labs!";

}

Play with these until you feel comfortable

# Some seasoning on the webapp

Ok but what if my application should expose more REST endpoints in the future? Or I want to run more webapp on my computer without any conflicts between them?

For solving the first problem let’s change the parameter of the @GetMapping annotation to “/greeting”. As new REST endpoints are added to the app, just come up with good paths for them.

For solving the other problems, let’s add a unique context path to our app and optionally change the port on which Tomcat listens to 8888. Open the application.properties file and insert the following lines:

server.servlet.context-path=/oalapp/services/onboarding

server.port=8888

From now you app can be accessed on the localhost:8888/oalapp/services/onboarding/greeting URL. For our real applications we always provide a specific context path, and it always starts with /oalapp/services.

Did you notice that application.properties file was empty so far and our app was still able to boot and run without any problem? And now we place 2 “random” properties into it and it not only remains functional but starts using the provided values. It is Springboot doing its magical autoconfiguration in the background, taking care of everything that you don’t want to, but immediately giving you the control if you ask for it.

# Our first unit test

Now let’s create some automated test for the code we have written so far. Below are some thoughts about writing automated tests. It might be too much information at this point but it has to exist in one piece somewhere. I recommend to come back to this later when more unit tests are written.

*It is required in the organization to cover the written code with automated tests as much as possible. It is not good only for testing the logic after it is implemented – automated tests can be executed any time to check if a new piece of code did not introduce regression.*

*Writing good automated tests starts with writing good code. Every time a new class or method is being designed, testability must be an important factor to think of. If tests are being written for a piece of code and it seems to be hard to do it correctly, it’s even worth restructuring the code if time allows.*

*There are basically 2 types of automated tests:*

1. *Unit test: focuses on 1 public method of 1 class. Classes should contain only a few public methods, most of the time 1 would be ideal. Unit tests are not written for private methods (it is not even possible with the standard tools), they will be indirectly involved into the test (because the public method calls them at a certain point). If the tested class invokes methods from other classes, there is 2 way to handle that:*
   * *Instantiate the called class in the test. Generally it is a good idea only for “data” classes, for example JPA entities*
   * *Have the testing framework create a mock object for the called class. Mocked objects have no real behavior, the author of the test defines what the methods of the class will return when invoked. Thus the developer has full control over the inputs of the tested method*
2. *Integration test: while unit tests instantiate only tested class, integration tests start the whole application. In case of Spring it means the whole application context is built. This takes some time, so integration tests are much slower compared to unit tests, but there are things that cannot be tested only with unit tests. Ideally an application should contain dozens, if not hundreds of unit tests, and only a couple of integration tests.*

*Note 1: it is possible to build just a slice of the app context which results in a test more complex than a unit test but still faster than an integration test. For example @WebMvcTest annotation is good for testing controllers because it just builds the web part of the app context but the tested controller receives a real HTTP request.*

*Although it is not always possible, the best scenario is when a class is covered 100% with tests. Above 80% is already very good. But it must be understood that not only the number of covered lines is important, but the branches too. Consider the following:*

if … {

huge logic

} else {

1 log message

}

*If only the “huge logic” branch is tested, it is possible that the line coverage is above 80% but the branch coverage will never be more than 50%! So write a test for the else branch too… or if it is hard to prepare a test case for that, the question is if the “else” branch even makes sense here.*

Below is an example test for my greeting controller:

package com.oracle.oal.microservices.onboarding;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;

import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

import org.junit.jupiter.api.Test;

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

import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;

import org.springframework.test.web.servlet.MockMvc;

import org.springframework.test.web.servlet.result.MockMvcResultMatchers;

@WebMvcTest

public class FirstControllerTestWebOnly {

@Autowired

private MockMvc mockMvc;

@Test

public void greetingForMissingName() throws Exception {

this.mockMvc.perform(get("/greeting"))

.andDo(print())

.andExpect(status().isOk())

.andExpect(content().string("Mysterious newcomer, welcome to Oracle Application Labs!"));

}

@Test

public void greetingForProvidedName() throws Exception {

this.mockMvc.perform(get("/greeting?name=Foo"))

.andDo(print())

.andExpect(status().isOk())

.andExpect(content().string("Foo, welcome to Oracle Application Labs!"));

}

}

# Setup a database

There is no real world application without a database, so let’s set up one for our application. Add the below dependencies to the pom.xml.

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

H2 is a fast, easy-to-use, lightweight in-memory database, rather for testing purposes than for production usage.

Capture some new properties to make H2 fully functional:

spring.datasource.url=jdbc:h2:mem:testdb;DB\_CLOSE\_DELAY=-1;DB\_CLOSE\_ON\_EXIT=false;MODE=ORACLE

spring.datasource.username=sa

spring.datasource.password=

spring.datasource.driver-class-name=org.h2.Driver

spring.h2.console.enabled=true

H2 console should be available on the http://localhost:8888/oalapp/services/onboarding/h2-console URL.

Spring JPA comes with Hibernate as default implementation, but in OAL Eclipselink is used instead. Follow [this guide](https://www.baeldung.com/spring-eclipselink) to replace Hibernate with Eclipselink and capture the following new properties:

spring.jpa.properties.eclipselink.weaving=false

spring.jpa.properties.eclipselink.ddl-generation=create-tables

spring.jpa.properties.eclipselink.logging.level.sql=FINEST

spring.jpa.properties.eclipselink.logging.parameters=true

# Our first JPA entity

JPA helps us manage relation data with OO tools. Usually the first step for managing our data is creating an entity class. Entities are POJOs representing a database table, each attribute linked to a column in the table.

Let’s create a JPA entity for managing the products of some company. First let’s create a new package structure for our JPA entities to have our code structured well:

com.oracle.oal.microservices.onboarding

|\_domain

|\_model

|\_product

In the lowest package create the below file

package com.oracle.oal.microservices.onboarding.domain.model.package;

import java.math.BigDecimal;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity // this makes this simple class a JPA entity

public class Product {

@Id

// id value will be automatically generated

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

private String productCode;

private String description;

private BigDecimal unitPrice;

private String uom;

}

Note: due to the representation issue of floating point types, we never use float or double for storing numbers requiring high precision (for example prices and amounts) but use BigDecimal instead.

Also create the following repository interface

package com.oracle.oal.microservices.onboarding.domain.model.product;

import org.springframework.data.repository.CrudRepository;

public interface ProductRepository extends CrudRepository<Product, Long> {

}

Spring repositories are interfaces org.springframework.data.repository.Repository<T,ID>, or one of it’s subinterfaces. During building the application, Spring will generate implementation classes for the interfaces and instantiates them as beans, so they can be injected to and used in any component.

If now the application is started and H2 console is open, there should be a table called PRODUCT generated with columns matching the entity attributes.

Note: in our real applications we never let JPA generate the db objects, but we create them before deploying the app. If JPA can see that a matching table already exists it uses that existing one instead of generating a new.

# Lombok

Add the following dependency to the pom.xml:

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

</dependency>

Lombok is a library which saves us time and makes the code more readable by providing simple annotations which in the background are transformed into real code. We can use the annotations instead of writing long pieces of repetitive code, for example getters for every attribute of a class.

# Seasoning the entity class

OAL has a naming convention for database objects than for Java variables, especially because database object names are always stored in uppercase, regardless of the format in the creation script. So \_ is supposed to be used between words in db object names (for example PRODUCT\_CODE). To solve the discrepancies between the 2 naming conventions we can use the javax.persistence.Table and javax.persistence.Column annotations to link entity attributes to columns with different name. Table annotation can also be used for listing indexes, Column annotation can also be used for specifying the length or the nullability of a column.

Exercise:

1. Use Table annotation on Product class to link the entity to a table with different name
2. Put an index on productCode attribute. (Note: OAL naming convention for indexes is <TABLE\_NAME>\_U1,U2… for unique indexes, <TABLE\_NAME>\_N1,N2… for non unique ones
3. Use Column annotation to link attributes to columns with name matching the OAL conventions
4. Also use the column annotation to specify different lengths for the String attributes
5. Verify that the values in the annotations are reflected in the generated H2 database too
6. Use Lombok annotations for generating getter methods, a builder (inner) class, a protected no-argument constructor and a private all-argument constructor

# Fully functional webapp with database

Add the following dependency to pom.xml:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-rest</artifactId>

</dependency>

Also add setter methods to the Product class with the help of Lombok.

Spring Data Rest automatically finds our Product model and the linked CRUD repository, and exposes the repository methods as REST endpoints based on the Hypertext Application Language standard. For example the URL for getting every product from the database in our application is (GET) <http://localhost:8888/oalapp/services/onboarding/products/>.

Exercise: launch the app and try out all the exposed endpoints.

So we have REST endpoints for querying, creating, modifying and deleting Products, all is reflected in the database – we have an app which theoretically could be deployed to a production environment to serve as the backend part of a product registry application.

# Enums as entity attributes

Let’s create an enum class in the …domain.product package called UnitOfMeasure with valid values EA, KG, METER. Use this enum as type for the uom attribute of the Product entity. Launch the application, use the appropriate POST endpoint to create a product and check the result in the database. The uom column contains a number. It’s not a bug, the application is able to parse this number into an enum instance, but it is inconvenient for example when a developer is debugging the app and checking the database. That’s why we always use the @Enumerated(EnumType.STRING) annotation on enum typed entity attributes.

Exercise: try out this annotation on the uom attribute

# Implementing business logic

Let’s imagine we have the following business requirement: from time to time our users want to adjust the prices of every product based on a minimum price. Probably in the future we need to be able to do it for a single product too.

Possible implementation:

package com.oracle.oal.microservices.onboarding.domain.model.product;

import java.math.BigDecimal;

import java.util.List;

import org.springframework.data.repository.CrudRepository;

public interface ProductRepository extends CrudRepository<Product, Long> {

List<Product> findByUnitPriceLessThan(BigDecimal minimumPrice);

}

package com.oracle.oal.microservices.onboarding.domain.model.product;

import java.math.BigDecimal;

import java.util.List;

import java.util.Optional;

public interface PriceAdjustmentService {

List<Product> adjustPriceForAllProducts(BigDecimal newPrice);

Optional<Product> adjustPriceForOneProduct(String productCode, BigDecimal newPrice);

}

package com.oracle.oal.microservices.onboarding.domain.model.product;

import java.math.BigDecimal;

import java.util.List;

import java.util.Optional;

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

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

@Service

public class PriceAdjustmentServiceImpl implements PriceAdjustmentService {

private final ProductRepository productRepository;

@Autowired

public PriceAdjustmentServiceImpl(ProductRepository productRepository) {

this.productRepository = productRepository;

}

@Transactional

public List<Product> adjustPriceForAllProducts(BigDecimal newPrice) {

List<Product> toProcess = productRepository.findByUnitPriceLessThan(newPrice);

toProcess.forEach(p -> p.setUnitPrice(newPrice));

return toProcess;

}

@Transactional

public Optional<Product> adjustPriceForOneProduct(String productCode, BigDecimal newPrice) {

Optional<Product> product = productRepository.findByProductCode(productCode)

.filter(p -> newPrice.compareTo(p.getUnitPrice()) > 0);

product.ifPresent(p -> p.setUnitPrice(newPrice));

return product;

}

}

First we need to be able to find products with unit price less than a given number. This is a pretty simple condition and if we put a method declaration with good enough name into our repository interface, Spring will be able to generate an implementation for this and ultimately generate the database query. Good enough name here means that it starts with “findBy” followed one condition (could me more), and condition is consists of a product attribute name (unitPrice) and an operator (lessThan).

Next we create an interface for our service (PriceAdjustmentService). Technically it would work if we created only the implementation class itself, but it is a good practice to do both: we can use the interface name everywhere in the code and replace the implementation class if needed without touching any other class (or even create multiple implementation for the same interface)

Finally we implement the interface in class PriceAdjustmentServiceImpl. A couple of things to take notice in this class:

1. Class is annotated with the org.springframework.stereotype.Service annotation. It means this class will be instantiated and the instance will be managed by Spring itself.
2. ProductRepository is also a Spring-managed component which can be automatically injected into our service, that’s what the Autowired annotation is for
3. Service attribute modifiers are important. We always follow the practice of declaring service attributes as private final. Thus the class instance will be immutable becoming a so called “stateless service”. Using only stateless services we can easily avoid concurrency problems without implementing any synchronization. This is very important. If it is not clear for you why it is needed or how to achieve it, contact your senior colleagues and ask for explanation.
4. Transactional annotation indicates that this code is running tied to a database transaction. This annotation is very important and using it improperly can lead to very hard bugs. Reading the documentation of it is inevitable, here are some important notes about it:
   1. Can be used on class and method level, method level is preferred
   2. Can be used only for public methods
   3. The db transaction ends only when the method ends, so if other methods are called they participate in the transaction even if they don’t have this annotation
   4. By default, if an annotated method is called and there is already a db transaction going on, it just joins the existing transaction
   5. It has parameters but most of the time they are not needed. The parameter most frequently used is the propagation = Propagation.REQUIRES\_NEW. It means the method starts a new transaction even if there is already one. It’s important, that the existing transaction does not end, only gets suspended until the new one ends
   6. If a not-annotated method calls an annotated one they must be in separate class otherwise no transaction is started
   7. At the end of the transaction, queried and modified entity instances are commited into the database unless exception occurred. New entities are not committed only if repository save method is invoked for them (read more about JPA entity lifecycle in documentation).   
      Invoking save method works for updated entities too, but it can lead to problems so not recommended.

And never forget writing unit tests for our services. Below is a possible test class, which contain true unit tests, no app context loaded, everything outside the service is mocked:

package com.oracle.oal.microservices.onboarding.domain.model.product;

import java.math.BigDecimal;

import java.util.ArrayList;

import java.util.List;

import java.util.Optional;

import org.junit.Assert;

import org.junit.Before;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.Mockito;

import org.mockito.junit.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.class)

public class PriceAdjustmentServiceTest {

@Mock

private ProductRepository productRepository;

@InjectMocks

private PriceAdjustmentServiceImpl priceAdjustmentService;

private BigDecimal newPrice = new BigDecimal(1000);

@Test

public void testPriceAdjustmentForAllProducts() {

List<Product> allProducts = new ArrayList<>();

allProducts.add(Product.builder().unitPrice(new BigDecimal(999)).build());

allProducts.add(Product.builder().unitPrice(new BigDecimal(998)).build());

Mockito.when(productRepository.findByUnitPriceLessThan(newPrice)).thenReturn(allProducts);

List<Product> result = priceAdjustmentService.adjustPriceForAllProducts(newPrice);

Assert.assertEquals(allProducts.size(), result.size());

for (Product product : result) {

Assert.assertTrue(newPrice.compareTo(product.getUnitPrice()) == 0);

}

}

@Test

public void testPriceAdjustmentForOneProductPositive() {

String productCode = "pr";

Product product = Product.builder().unitPrice(new BigDecimal(999)).build();

Mockito.when(productRepository.findByProductCode(productCode)).thenReturn(Optional.of(product));

Assert.assertTrue(priceAdjustmentService.adjustPriceForOneProduct(productCode, newPrice).isPresent());

Assert.assertTrue(newPrice.compareTo(product.getUnitPrice()) == 0);

}

@Test

public void testPriceAdjustmentForOneProductNotInDb() {

String productCode = "pr";

Mockito.when(productRepository.findByProductCode(productCode)).thenReturn(Optional.empty());

Assert.assertFalse(priceAdjustmentService.adjustPriceForOneProduct(productCode, newPrice).isPresent());

}

@Test

public void testPriceAdjustmentForOneProductPriceNotLow() {

String productCode = "pr";

Mockito.when(productRepository.findByProductCode(productCode)).thenReturn(Optional.of(

Product.builder().unitPrice(new BigDecimal(1000)).build()));

Assert.assertFalse(priceAdjustmentService.adjustPriceForOneProduct(productCode, newPrice).isPresent());

}

}

The new service can be wired into some triggering component, for example a simple controller below (notice the package difference between the service and the controller):

package com.oracle.oal.microservices.onboarding.application.product;

import com.oracle.oal.microservices.onboarding.domain.model.product.PriceAdjustmentService;

import com.oracle.oal.microservices.onboarding.domain.model.product.Product;

import java.util.List;

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

import org.springframework.web.bind.annotation.PostMapping;

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

@RestController

public class PriceAdjustmentController {

private final PriceAdjustmentService priceAdjustmentService;

@Autowired

public PriceAdjustmentController(PriceAdjustmentService priceAdjustmentService) {

this.priceAdjustmentService = priceAdjustmentService;

}

@PostMapping("/products/priceadjustment")

public List<Product> priceAdjustment(@RequestParam BigDecimal newPrice) {

return priceAdjustmentService.adjustPriceForAllProducts(newPrice);

}

}

Exercise: use an HTTP client to validate the functionality developed

# Getting started with Apache Kafka

Almost all the communication between the micro services developed by our org happens through Apache Kafka.

Exercise: install Apache Kafka on your laptop (for example following [this page](https://kafka.apache.org/quickstart)). Make sure you are able to launch the Zookeper and the Kafka server itself

Add the following dependency to pom.xml:

<dependency>

<groupId>org.springframework.kafka</groupId>

<artifactId>spring-kafka</artifactId>

</dependency>

Then add the following property to the property file: spring.kafka.bootstrap-servers=localhost:9092

Basically these 2 steps give you a fully functional Kafka client with a ton of default configuration. During consuming and producing messages we would be able to work with simple String format messages. However most of the time we want to parse the messages into various data structures. There are several ways to achieve this, the common approach in our team is letting the Kafka client itself do the parse automatically so we don’t need to write boilerplate code. For this we need to tell the Kafka clients (listener and producer) to use for example Json converters on the messages. At this point we need to introduce a Java-based Kafka config file:

package com.oracle.oal.microservices.onboarding.infrastructure;

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

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.kafka.annotation.EnableKafka;

import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;

import org.springframework.kafka.core.ConsumerFactory;

import org.springframework.kafka.core.KafkaTemplate;

import org.springframework.kafka.core.ProducerFactory;

import org.springframework.kafka.listener.ErrorHandler;

import org.springframework.kafka.support.converter.StringJsonMessageConverter;

@Configuration

@EnableKafka

public class KafkaConfig {

private final ErrorHandler errorHandler;

/\*\*

\* Initialize Kafka Config

\*

\* @param errorHandler err handler

\*/

@Autowired

public KafkaConfig(ErrorHandler errorHandler) {

this.errorHandler = errorHandler;

}

/\*\*

\* Override the default ContainerFactory to set custom configuration

\*

\* @param consumerFactory request factory

\* @return ConcurrentKafkaListenerContainerFactory container factory

\*/

@Bean

public ConcurrentKafkaListenerContainerFactory<String, String> kafkaListenerContainerFactory(

ConsumerFactory<String, String> consumerFactory) {

ConcurrentKafkaListenerContainerFactory<String, String> factory =

new ConcurrentKafkaListenerContainerFactory<>();

factory.setConsumerFactory(consumerFactory);

factory.setMessageConverter(new StringJsonMessageConverter());

factory.setErrorHandler(errorHandler);

return factory;

}

/\*\*

\* Override the default KafkaTemplate to set custom configuration

\*

\* @param producerFactory producer factory

\* @return KafkaTemplate

\*/

@Bean

public KafkaTemplate<String, String> kafkaTemplate(ProducerFactory<String, String> producerFactory) {

KafkaTemplate<String, String> kafkaTemplate = new KafkaTemplate<>(producerFactory);

kafkaTemplate.setMessageConverter(new StringJsonMessageConverter());

return kafkaTemplate;

}

}

Converting simple Strings into Json structures during consumption can fail and Spring needs to know what to do when it happens. So we need to implement at least one error handler. The below example is the simplest error handler one can imagine, does nothing but logs the error. It is sufficient for us and actually you can find the same implementation in many of our real services.

package com.oracle.oal.microservices.onboarding.infrastructure;

import lombok.extern.slf4j.Slf4j;

import org.apache.kafka.clients.consumer.ConsumerRecord;

import org.springframework.kafka.listener.ErrorHandler;

import org.springframework.stereotype.Component;

@Component

@Slf4j

public class DefaultKafkaErrorHandler implements ErrorHandler {

@Override

public void handle(Exception thrownException, ConsumerRecord<?, ?> data) {

log.error("Error processing message from Topic: {}, Partition: {}, Offset: {}, Timestamp: {}.", data.topic(),data.partition(), data.offset(), data.timestamp(), thrownException);

}

}

# Consuming data from Kafka topic

A client program which is able to pull messages from a Kafka topic (more precisely a partition of a topic) is called a consumer. A consumer is a big and complex client, which regularly polls the Kafka broker for new messages, commits the offset after successful consumption, periodically sends heartbeat, and does many other thing. Spring Kafka hides this all from developers by adding an additional abstraction on the consumer, called listener. As opposed to consumers, listeners need only a few config value to be fully functional.

Let’s suppose an external system wants to notify our service if a product needs a new price. The system sends Kafka messages like this one:

{"productCode" : "pr1", "price" : 1000.0}

We have to be able to parse this messages into data structure understandable by our Java code, so let’s define the below class:

package com.oracle.oal.microservices.onboarding.application.product;

import java.math.BigDecimal;

import lombok.AccessLevel;

import lombok.AllArgsConstructor;

import lombok.Getter;

import lombok.NoArgsConstructor;

import lombok.Setter;

import lombok.ToString;

@ToString

@AllArgsConstructor

@NoArgsConstructor

@Getter

public class PriceAdjustmentMessageEvent {

private String productCode;

private BigDecimal price;

}

If we receive such a message, we can simply invoke the appropriate method of the service implemented earlier. The full listener code can be as simple as below:

package com.oracle.oal.microservices.onboarding.application.product;

import com.oracle.oal.microservices.onboarding.domain.model.product.PriceAdjustmentService;

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

import org.springframework.kafka.annotation.KafkaListener;

import org.springframework.stereotype.Service;

@Service

public class PriceAdjustmentListener {

private PriceAdjustmentService priceAdjustmentService;

@Autowired

public PriceAdjustmentListener(PriceAdjustmentService priceAdjustmentService) {

this.priceAdjustmentService = priceAdjustmentService;

}

@KafkaListener(id = "priceAdjustmentListner", groupId = "onboardingGroup", topics = "priceAdjustmentTopic")

public void consumePriceAdjustmentMessageEvent(PriceAdjustmentMessageEvent event) {

priceAdjustmentService.adjustPriceForOneProduct(event.getProductCode(), event.getPrice());

}

}

Listener is a method of a regular service annotated with @KafkaListener. Listener has an id, a group (make sure you understand the concept of Kafka consumer groups) and it can listen to one or more topics (usually we make them listen only one). In this example all these are hard-coded string values, but in real services they come from config property. For example if we have a property in the file like onboarding.priceadjustment.topic=priceAdjustmentTopic the annotation parameter can be   
topics = ${onboarding.priceadjustment.topic}.

Exercise: move topic and groupId to config property

Notice the parameter type of the method being PriceAdjustmentMessageEvent. Here is roughly how our service gets a PriceAdjustmentMessageEvent from Kafka:

1. The consumer polls the Kafka broker and if there are new messages available, it receives a batch of them.
2. The messages are in binary format and consumer deserialize them as String
3. For every element of the batch: based on the listener method parameter and the listener config Spring converts the String messages into Json structure and then into a matching message event type, which is passed to the listener method.

# Publishing Kafka messages

Just like listener was created on top of consumer, a new layer is added on the top of publisher too, which is called Kafka Template, which can be autowired into any service. With the help of template it is very easy to publish messages: we just create a message with a payload and at least one header value (the Kafka topic) and pass it to the template. For example the below service can publish Products:

package com.oracle.oal.microservices.onboarding.application.product;

import com.oracle.oal.microservices.onboarding.domain.model.product.Product;

import org.springframework.kafka.core.KafkaTemplate;

import org.springframework.kafka.support.KafkaHeaders;

import org.springframework.messaging.Message;

import org.springframework.messaging.support.MessageBuilder;

import org.springframework.stereotype.Service;

@Service

public class ProductPublisher {

private final KafkaTemplate<String, String> kafkaTemplate;

public ProductPublisher(KafkaTemplate<String, String> kafkaTemplate) {

this.kafkaTemplate = kafkaTemplate;

}

public void publish(Product product) {

Message<Product> message =

MessageBuilder.withPayload(product).setHeader(KafkaHeaders.TOPIC, "productRepriced").build();

kafkaTemplate.send(message);

}

}

Now let’s autowire ProductPublisher into PriceAdjustmentServiceImpl and modify the adjustPriceForOneProduct method as below:

@Transactional

public Optional<Product> adjustPriceForOneProduct(String productCode, BigDecimal newPrice) {

Optional<Product> product = productRepository.findByProductCode(productCode)

.filter(p -> newPrice.compareTo(p.getUnitPrice()) > 0);

product.ifPresent(p -> {

p.setUnitPrice(newPrice);

productPublisher.publish(p);

});

return product;

}

Now assuming there is already a product in the database, and for that product a price adjustment message is received by our program, it not only updates the price but publishes a latest version of the product to Kafka.

Exercise:

1. Start the application
2. Create a product with the appropriate endpoint exposed by Spring Data Rest
3. Start a Kafka producer using the kafka-console-producer script for priceAdjustmentTopic
4. Start a Kafka consumer using the kafka-console-consumer script for productRepriced topic
5. Produce a message to priceAdjustmentTopic
6. Verify that the product is updated in the database and the consumer consumed the message about the fresh product

# Logging

For logging we use Slf4j. Slf4j is an abstract logging API which needs to be backed by a logger implementation. Fortunately our Springboot app already has such an implementation called Log4j2. The only thing we need to do is putting the @Slf4j Lombok annotation on any class, and we can use the log.trace, log.debug, log.info, log.warn and log.error methods to generate log messages. The default Log4j2 config channels the log messages to the standard output.

Exercise: make sure you understand logging levels

Logging levels can be changed by setting the logging.level.root=<log level> property (for all packages) or the logging.level.<package>=<log level> (for the specified package).

# Kafka publish result

Publishing messages with Spring Kafka is an asynchronous process, it does not happen immediately when the send method of the KafkaTemplate is invoked, the underlying publisher takes care of it based on its own configuration. That’s why the result type of the send method is a ListenableFuture. “java.util.Future” interface is part of core Java API and it is used for managing an asynchronous computation. ListenableFuture is a Spring interface extending Future. It allows developers to register callbacks which are invoked by the async thread based on the outcome of the performed computation.

Let’s modify our publish method:

public void publish(Product product) {

Message<Product> message =

MessageBuilder.withPayload(product).setHeader(KafkaHeaders.TOPIC, "productRepriced").build();

ListenableFuture<SendResult<String, String>> sendFuture = kafkaTemplate.send(message);

sendFuture.addCallback(new ListenableFutureCallback<SendResult<String, String>>() {

@Override

public void onFailure(Throwable throwable) {

log.error("Error publishing repriced event for product {}", product.getProductCode(), throwable);

}

@Override

public void onSuccess(SendResult<String, String> stringStringSendResult) {

log.info("Succesfully published repriced event for product {} to topic {}",

product.getProductCode(), stringStringSendResult.getProducerRecord().topic());

}

});

}

Notice that in the callback methods we have access to the variables used in the publish method, so we can do more than just logging the result. For example if we had a status attribute in the Product model, we could update it with SUCCESS or ERROR values depending on which callback method was invoked.

# Spring Kafka under the hood

Exercise:

1. Launch the application
2. Once started, look for the “ConsumerConfig values” text in the logs, and examine the key-value list which below that
3. Insert one product into the database either with direct sql commands or the appropriate REST call
4. Using Kafka console produce a price adjustment message applicable for the existing product
5. Once the repriced message event is published, look for the “ProducerConfig values” text in the logs and examine the key-value list below that

The two huge lists contain the configuration values for the consumer and the producer respectively. As mentioned earlier, consumer and producer are the client components communicating with Kafka brokers, listener and Kafka template are just abstraction to make our job easier.

But where do those config values come from? They are automatically set by Spring Kafka. Fortunately for most of our tasks this predefined configuration works well and while everything works it is not a good idea to touch it. But for any special requirements that may arise in the future, it is important to be aware that there is a way to modify this configuration. And how to do that? In our KafkaConfig class created earlier, both factories (for consumer and for producer) accept Maps of properties. If interested more, read Spring Kafka documentation.

# Event driven development

Spring provides out-of-the-box toolset for publishing and consuming application events.

Let’s modify the PriceAdjustmentServiceImpl class:

1. Add a new attribute private final ApplicationEventPublisher applicationEventPublisher. It is a Spring-managed bean and can be initialized in the constructor
2. Remove the ProductPublisher attribute
3. Replace the productPublisher.publish(p); line with applicationEventPublisher.publishEvent(p).

Then put the @EventListener annotation on the ProductPublisher.publish method. If the application is launched now it works perfectly well, we easily replaced the direct method invocation with publishing an event about a repriced product and catching the same event. But why would we do this? Imagine the following situations:

1. We want to switch from Kafka to another messaging system so we need a new publisher. As the Kafka publisher is no longer wired into the reprice service we don’t need to modify and retest that service, just create the new publisher
2. In addition to publishing to a message broker we want to send an automatic email notification about the repriced product. And later save the reprice event to a separate database table for reporting purposes. And later…   
   Instead of adding more and more component to the reprice service (and retesting it every time) we just create new classes with methods listening to the same event and we are done

So events can be very useful, but it does not mean it makes sense replacing each and every direct method invocation with events.

Note: let’s imagine we get more and more feature requests and our application grows bigger and bigger. At some point we need to publish events about a product getting updated description. Now if we just publish a Product as an application event, the ProductPublisher will catch the event and do the Kafka publish. But what if we want Products to be published to Kafka only if they are repriced…?   
Based on this thought process we need to realize that publishing entities, dto-s or similar objects as events is not a good idea (we just did it for the sake of simplicity). Instead let’s always create a separate event class, for example a ProductRepricedEvent here. Events are just simple POJOs (with earlier Spring versions they had to extend a certain class but now they don’t), but what attribute they should hold? For our situation it is possible to simply have a Product attribute, so a ProductRepricedEvent instance carries the Product instance that was actually repriced (code would be something like applicationEventPublisher.publish(new ProductRepricedEvent(p);). This would work for most of the time, but can cause strange bugs if database transactions are involved into the situation too. So for entities the best if the ID of the entity is travelling in the events, and the event listener queries the entity from its repository using the ID.