Coding Task (Feb 2022)

Introduction

The aim of this coding task is to see how you go about solving a problem given a set of basic requirements. There are no right or wrong answers, but successful candidates will most likely have followed good coding standards and will have tested their solution somehow. We don't expect a production-ready solution just that you can demonstrate your skills to us.

Good luck!

Instructions

Follow the steps below to complete the coding task. The initial User Story should give you some background context. Unless specified otherwise, you can create suitable names for endpoints, classes and methods as required.

User Story

As a Maersk customer
I want to be able to book containers with Maersk
So that I can deliver cargo to my customers

The aim of this story is to develop two microservice endpoints that enable a customer to book a container with Maersk.

- **There is no need to consider authentication or authorization mechanisms for this task.**

- **There is no need to consider the UI as this is a purely backend (API related) task.**

One endpoint (endpoint 1) will establish if there are enough containers of an appropriate size and type at a given container yard to meet the customers booking requirements. The service acts as a proxy and will call another external service (which doesn't really exist) to fetch the data.

The other endpoint (endpoint 2) will receive a booking request from the customer and store the data in a Cassandra database table for later processing by other systems.

Steps

1. Create a new reactive (Web Flux) Spring Boot project using Java 11. Dependencies can include whatever you feel appropriate but, as a minimum, must include the following dependencies:

```xml
<dependencies>
	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-data-cassandra-reactive</artifactId>
	</dependency>
	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-webflux</artifactId>
	</dependency>

	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-test</artifactId>
		<scope>test</scope>
		<exclusions>
			<exclusion>
				<groupId>org.junit.vintage</groupId>
				<artifactId>junit-vintage-engine</artifactId>
			</exclusion>
		</exclusions>
	</dependency>
	<dependency>
		<groupId>io.projectreactor</groupId>
		<artifactId>reactor-test</artifactId>
		<scope>test</scope>
	</dependency>
</dependencies>
```

2. Create a new POST endpoint (endpoint 1). There are two ways to do this (annotated controllers or functional endpoints) so please choose whichever way you prefer. The endpoint should be callable from the root context of "/api/bookings/" e.g. the url would look like: "localhost:8080/api/bookings".
    a. Your endpoint will call an internal service (that you create within the same project) that will itself call an external service (that doesn't really exist so you'll need to mock it!) at endpoint: https://maersk.com/api/bookings/checkAvailable
    b. The external endpoint will return a JSON object in the form of a key called "availableSpace" and an integer value. For example:

    ```
    {
      "availableSpace" : 6
    }
    ```

    c. If the answer from the external "checkAvailable" service is 0 (zero), you should return:

Classification: Public

```
{
  "available": false
}
```

d. Otherwise you should return:

```
{
  "available": true
}
```

e. The object that your POST endpoint (endpoint 1) should expect to receive from the customer (consumer) will be in the form of:

Key	**Value constraints**
containerSize	Integer – either 20 or 40
containerType	Enum – DRY, REEFER
origin	String – min 5, max 20
destination	String – min 5, max 20
quantity	Integer – min 1, max 100

Example:

```
{
  "containerType" : "DRY",
  "containerSize" : 20,
  "origin" : "Southampton",
  "destination" : "Singapore",
  "quantity" : 5
}
```

f. Your endpoint (endpoint 1) should validate the input in whichever way you think best.

3. Create a second POST endpoint (endpoint 2). The endpoint should be callable from the root context of "/api/bookings/". Your endpoint will return a JSON object in the form of a key called "bookingRef" and a String value which will be a UUID. For example:

```
{
  "bookingRef" : "94558c91-f344-4ddd-b160-2c64625f7d29"
}
```

a. Your endpoint (endpoint 2) will call an internal service (that you create in the same project) that will store the data received in the request to a Cassandra table called “bookings”. The keyspace name should be called “maersk”. Cassandra configuration should be via the application.yaml file. The names of Cassandra columns should be in snake_case format. The UUID discussed above will be the primary key to the table.

b. If the data is correctly saved, you should return an object with the key of “bookingRef” and the key of the Cassandra record as the value. For example:

```
{
  “bookingRef”: “94558c91-f344-4ddd-b160-2c64625f7d29”
}
```

c. The API should NOT expose any Spring or Cassandra related internal exceptions to the consumer (think about @ContollerAdvice) and should instead return a plain “HTTP 500 INTERNAL SERVER ERROR”. The exceptions should however be logged for future investigation.

d. The object that your POST endpoint should expect to receive from the consumer will be in the form of:

Key	**Value constraints**
containerSize	Integer – either 20 or 40
containerType	Enum – DRY, REEFER
origin	String – min 5, max 20
destination	String – min 5, max 20
quantity	Integer – min 1, max 100
timestamp	String - ISO-8601 date and time for UTC timezone e.g. 2020-10-12T13:53:09Z

Example:

```
{
  “containerType” : “DRY”,
  “containerSize” : 20,
  “origin” : “Southampton”,
  “destination” : “Singapore”,
  “quantity” : 5,
  “timestamp” : “2020-10-12T13:53:09Z”
}
```

e. Your endpoint should validate the input in whichever way you think best.

4. Your solution should be tested in whichever way you think appropriate. As a minimum, the calls to endpoint 1 and endpoint 2 should be tested.