Coding Task

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, produced self-documenting and clean code and will have tested their solution somehow.

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.

One endpoint 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 to fetch the data.

The other endpoint will receive a booking request 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:

```
<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. There are several ways to do this so please choose whichever way you think most appropriate. The endpoint should be callable from the root context of "/api/bookings/".
    a. Your endpoint will call an internal service (that you create) that will itself call an external service (that doesn't really exist!) 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 this "checkAvailable" service is 0 (zero), you should return:
    ```
    {
      "available": false
    }
    ```
    d. Otherwise you should return:
    ```
    {
      "available": true
    ```

```
}
```

e. 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

Example:

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

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

3. Create a second POST endpoint. 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 number in the form of "957xxxxxx", so 9 digits in total. The number will start at 957000001 and will increment 1 for every record saved in the database. For example:

```
{
  "bookingRef" : "957000002"
}
```

a. Your endpoint will call an internal service (that you create) that will store the data received in the request to a Cassandra table called "bookings". The keyspace name should be domain related. Cassandra configuration should be via an appropriate yaml file. The names of Cassandra columns should be in snake_case format. The number discussed above (957xxxxxx) 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": "957000002"
}
```

c. The API should NOT expose any Cassandra internal exceptions to the consumer and should instead return an "INTERNAL SERVER ERROR" with the message "Sorry there was a problem processing your request". 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.