Coding process for the Activities Test

First step: API definition

The first I did once I finished reading the test requirements was to define the service, a REST service. I defined the API using an API online editor, in the last project I was using *Apiary* but I steel feel more comfortable with *Swagger*, so I opted for that one. I knew that maybe this will mean to dedicate some extra time before start coding but I think investing time to define first the API helps a lot not only later when you are coding but also to understand correctly the requirements.

Although the test works only for Madrid since the beginning I wanted to introduce the parameter for the *city* as is mentioned in the requirements. I also defined in that point the objects to wrap the data to be managed:

- Activity as base object to manage the data of an activity
- OpeningHours to manage the available time schedule of each activity
- Point: to define the coordinates of the activity

The first endpoint to retrieve all the activities based on a provided filters was quite straightforward to define. Nothing special to mention.

The second endpoint, to retrieve a recommended activity was different. I had some doubts regarding how to manage the parameters for starting and ending times. Finally I decided to do it simple and require, apart from the city, only a datetime to define the start time, and a datetime to define the end time. The only difference will be on the times of the parameters, as the dates will be the same (at least in this first approach) but I thought that is more flexible to do it in that way, so maybe in the future the functionality could be extended for more days etc.

The API definition can be found in the *activities.yaml* file included in the repo.

Infrastructure and scaffolding

Obviously since the beginning I decided to develop the test using the programming language that I use on a daily basis nowadays. That language is Java.
Regarding the frameworks and tools, the same approach, I have used what I use daily:

- Spring Boot as Java framework.
- Maven as dependency manager.
- Intellij as IDE.
- Git for version control.

Although Swagger provides an option to generate the server app structure based on the API definition I have created the Spring Boot project using the Spring Initializr (https://start.spring.io/) service. Doing it in that way you have the latest stable versions of the Spring framework.

Architecture

Although the test doesn't have access to a real db I opted for the typical three-layers solution:

- service: to store the REST definition and implementation.
- logic: to locate the business logic of the service and the data transfer objects.
- dataaccess: to put the classes that read the json file with the activities data.

In a bigger app I would have divided the packages in different components to have the code clearly separated:

- APP
    - Activity management component
        - service
        - logic
        - dataaccess
    - Other component
        - service
        - logic
        - dataaccess
    - ...

But in this case, as the service is so simple, I compressed it directly in a root package *activities* that contains the other three packages:

- Activities APP
    - service
    - logic
    - dataaccess

Dependency Injection

I have used the basic dependency injection that Spring provides: https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-spring-beans-and-dependency-injection.html
For that reason the *logic* and the *dataaccess* layers contain two packages: *api* and *impl*

- api: includes the interface
- impl: has the implementation of the interface

Implementation

For the implementation I normally work with TDD so I started writing the first tests (*ActivitiesApplicationTests.java*). Then to pass the tests I found the critical points for me in this service:

- Read the json
- Apply the filters

My first idea was to convert the json to a list of Java objects and apply the filters using the Java's lambda expressions over that list. I knew that I didn't want to deal with parsing the json file, mapping or transforming it into Java objects, and list and so forth. So I took a look to the available libraries and found Gson (https://github.com/google/gson). I didn't know this library but fortunately it has worked really well :)

So finally, for reading the json file I decided to use the *resources* directory that Spring provides. Doing it in that way the app can easily access to the content of the json file with two lines of code and the json file is included in the package of the app when creating the artifact. You can check that the *madrid.json* is located in the */activities/src/main/resources* directory.

With this two solutions creating a list of objects with the activities was quite straightforward.

Data access layer

I decided to put here only the reading process of the json. So the filters will be applied as a business logic part in the logic layer.

Reading the json from the *resources* directory and using the *Gson* library did the work.

Logic layer

The logic for retrieving and filtering the activities based on *category*, *location* and *district* wasn't complicated. I used the lambda expressions of Java and once the list of activities was retrieved the filtering was quite easy to complete.
The logic to find the best activity, filtering based on time ranges, was different. It was not so easy. I decided to work with four dates:

- *targetStart*: when the user wants to start the activity
- *targetEnd*: when the user wants to end the activity
- *activityStart*: when an activity actually starts
- *activityEnd*: when an activity actually ends.

As mentioned in the requirements the valid activity is the one that "fits in the time range", so I simplified it to two conditions.

- *targetStart* is equals or after *activityStart*.
- *targetEnd* is equals or before *activityEnd*.

The *targetStart* and *targetEnd* are parameters provided in the call so I needed to obtain the *activityStart* and *activityEnd* dates, for each activity. What I did was after filtering by category (reusing the method I created to apply the filters of the *get-all-activities* service) I obtained for each activity the correct day of the week, got the opening hours of the day, and generate the activityStart and activityEnd dates based on that opening hours range.

Finally I discarded activities based on the mentioned two conditions. And that's basically all.

If more than one activity fit the time range I apply a filter to obtain the activity with a higher *hours_spent* value, as specified in the requirements.

Service layer

Nothing to mention here, I used the standard REST implementation that Spring provides for Java applications. I think the only thing important in this layer is to avoid including logic. So it only contains the definition of the REST endpoints.

Other considerations

Finally, regarding how to extend the program, some quick thoughts:

- *Do not recommend an outdoors activity on a rainy day* → As a new filter we could simply call an external weather service (https://openweathermap.org/, http://apidev.accuweather.com/developers/) to get the forecast for the day and hour of the activity. In case of risk of rain for that time we could simply apply the *indoor* location filter.
- *Support getting information about activities in multiple cities* → as the service has the approach for filtering by city we could create a new service that uses the current one and gets the information for different cities. Even we could get a list of sequential time schedules of activities in different cities.
- *Extend the recommendation API to fill the given time range with multiple activities* → this could be a quite large functionality with multiple approaches but, as we know the time spent in each activity maybe we could fill the time range based on that data, the location of the activities, and calculating the time needed for the transport between locations (with traffic forecasts or some similar consideration). Also the weather condition could be applied, in case of rain or bad weather in general the service could provide a complete route through all indoor activities ordered by opening hours.