ILP CW2 Specifications (Programming Task)

25.10.2024

The first assignment has been designed as a preliminary to tackling larger and more complex situations in the second assignment.

Your main tasks can be summarized as follows:

- Extend the Java-REST-Service built in CW1
  - Must be in Java
  - Implement the necessary endpoints for the POST and GET requests (see below)
  - Proper parameter handling (check for invalid data) – see below
  - Proper return code handling – see below
  - JSON handling

- Place the service in a docker image
  - **amd64** as target architecture – **not arm64** (this is relevant for Mac users!)
  - you can build a cross-platform image using the ***docker buildx build*** command with the –platform option (for details see https://docs.docker.com/reference/cli/docker/buildx/build/)

- save the docker image in a file called **ilp_submission_image.tar**
  (it is in TAR format anyhow)

- **place** the file **ilp_submission_image.tar into your root directory of your solution**

  Your directory would look something like this:

  ```
  ilp_submission_2 (your root directory)
      ilp_submission_image.tar
      src (the Java sources…)
          main
                  …
          …
  ```

- Create a ZIP file of your solution directory
  - Image
  - Sources
  - IntelliJ (or whatever IDE you are using) project files

- upload the ZIP as your submission in Learn

The REST-Service must provide the following endpoints:

1. All entry points as defined by CW1 in a correct form. You might have to fix something depending on the feedback you will receive for CW1.

   These entry points will be tested briefly again with lower points for them

2. `validateOrder` (POST)

   return the `OrderValidationResult` for the order in the body.

   The validation result is defined in chapter 1.6 of the ILP 2024 global document and the various possible codes in 1.6.4.

   The `Order` is defined in 1.6.1

   This method will be called with valid and invalid data, yet always an order (so no empty calls). An order might contain more data which you can ignore – so, you are only using the defined order.

   As each order passed in is an order (only some fields might have wrong values, etc.) you are supposed to return a result (and 200). Any other code will be considered invalid.

   This endpoint will be called with a single error constellation to test for. So, no double errors (like invalid pizza count and CVV invalid).

   Given the high points for the error checks, please make sure you identify them properly.

   To help you with that you can retrieve a list of orders including an orderStatus and orderValidationCode using https://ilp-rest-2024.azurewebsites.net/orders. This list will contain all possible errors and good data as well.

3. **`calcDeliveryPath (POST)`**

   given a valid order (others shall return 400) return the path (including hovers) from the restaurant in the order to AT avoiding no-fly-zones and once inside the central area not leaving it again.

   In the request the body will contain the Order and given a proper order the response is supposed to be an array of LngLat (read about hovering and positions in the ILP document)

```
[{
  "lng": -3.192473,
    "lat": 55.946233

  }]
```

This method will be called with valid and invalid data (semantically and syntactically). Only for proper syntactically correct records, you are supposed to return a result (and 200); otherwise, you shall return 400 (bad request).

You will be marked for the correctness (and adherence of the spec) as well as your speed of implementation and speed of the drone course as such.

For the speed tests all student submissions are recorded with the msec it took to return the result, and the flight path segments (given they were correct). This is then grouped in 4 buckets with marks from 1..4 for each speed category (or 0 if a path was not correct). This way, it might happen that you are perhaps in the group of the fastest runtime, yet your path might be not as good, etc.

4. **calcDeliveryPathAsGeoJson (POST)**

   This endpoint shall return the same flightpath as the result for calcDeliveryPath, just in GeoJSON-format **where you omit the hovering steps**.

   If the result is properly formatted and returned, you should be able to paste the resulting GeoJSON structure in: https://geojson.io/#map=2/0/20 and see the visual results.

   Only valid orders will be passed to this endpoint and the same grouping for speed as previously described is performed.

   For more information on the GeoJSON-format have a look here: https://en.wikipedia.org/wiki/GeoJSON or the official spec: https://datatracker.ietf.org/doc/html/rfc7946

   This endpoint is intended as an advanced implementation feature and should only be attempted, if you mastered the other endpoints as you will need the proper handling of those for this to function.

All data necessary for POST-endpoints will be passed in the body of the request as a JSON data structure and any results will be returned there too.

To help you with your assignment, there is a GitHub repository, where all JSON data is shown exemplary as well as implementations of the data classes.
Have a look: https://github.com/mglienecke/IlpDataObjects

The following should be considered when implementing the REST-service:

- Do proper checking for URLs, data, etc. Don't handle anything not accurate (you will receive error data and requests!)
- You are using http, not https
- Your endpoint names must match the specification (so **no global prefix** like /api, /ilp, etc.). Your API must be reachable at i.e. **http://server:8080/uuid**
- Test your endpoints using a tool like Postman or curl. Plain Chrome / Firefox, etc. will do equally for the GET operations
- The filename for the docker image file must be exactly as defined as well as the location of it in the ZIP-file. Should you be in doubt, use copy & paste to get the name right

- **You can use any library and implementation detail you want if the final image is runnable in docker and uses Java**

Should you need help:

- See the literature links in the "Library" section in Learn. You should find most information there
- If you cannot find an answer to your question, please post it on Piazza, though try finding it yourself first, please (as we have only limited capacity)

Disclaimer: We will not be able to answer any question on piazza less than 3 days before the assignment deadline
So, please make sure you start the assignment in good time.

Marking:

This programming task (CW2) has a maximum mark of 45 / 100 points in relation to the entire ILP course (25/100 will be awarded for the essay you will provide as part of the CW2 submission).

From the 45 maximal points, 80% (36 points) will be auto marked (like CW1) and 20% (9 points) for manual checks of your source code.

The manual mark breakdown will be:

- Code Quality 4 points
  usage of comments, class layout, source structure, variables, etc.

- Unit testing 5 points
  MVC-tests, unit tests, mocking, etc.

The auto marked points (36) will be purely allocated on auto-tests with the following values:

Endpoint	**Comment**	**Points**
Re-running existing tests (4)	Correct call	4
validateOrder (10)	Correct call	3
	Various error constellations (correct structure, but invalid data)	7
calcDeliveryPath (16)	Correct call	6
	Call with logical errors (order validation)	2
	Computational speed to return results relative to all others.	4
	Path efficiency (moves required)	4
calcDeliveryPathAsGeoJson (6)	Correct call	4
	Efficiency	2

We cannot build docker images for your source codes. Should you fail to provide a runnable docker image according to the specification or provide no source code in the submission, no marking will be possible, and you will receive 0 points.

Please make sure your image can be loaded into docker, is runnable and reacts to curl (or postman) requests