

---

# Informatics Large Practical

Michael Glienecke

School of Informatics, University of Edinburgh

**This document is "work in progress".  
Please consider it as a guide and idea, not a final version**

Document version 4.1.0

---

## About

The Informatics Large Practical is a 20 point Level 9 course which is available for Year 3 undergraduate students on Informatics degrees. It is not available to visiting undergraduate students or students in Year 4 or Year 5 of their undergraduate studies. It is not available to postgraduate students. Year 4, Year 5 and postgraduate students have other practical courses which are provided for them.

## Scope

The Informatics Large Practical (ILP) is an individual practical exercise which consists of one large design and implementation project, with two coursework submissions.

The **general aim** of ILP is to familiarize you with current state-of-the-art techniques and methodologies used in software engineering as well as providing you with a sound foundation for your own future software engineering work. To achieve this, you will have lectures where more general information is taught (including some topics you have had in the past as a refresher), and lectures where more programming related things are discussed.

As usual, the lectures serve as information provisioning, whereas tutorials and Q&A are there to help you achieve your course works and explain some topics in more detail.

**Coursework 1 (CW1)** involves creating a new project to implement a basic REST-service with limited functionality, running inside docker as a container. This service serves as a foundation for the second assignment. *The main focus is to make sure you have the necessary foundational knowledge for assignment 2 as well as a basic understanding of the docker container context and REST services*

**Coursework 2 (CW2)** is the implementation of the entire project together with a small report on the implementation and decisions made. *The main focus is on programming and implementing your solution*

*Please note that the two coursework are not equally weighted. There is no exam paper for ILP so, to calculate your final mark out of 100 just add together your marks for the coursework.*

| Coursework   | Weight | Consists of                                  |
|--------------|--------|--|
| Coursework 1 | 30%    | programming task (all of the mark)           |
| Coursework 2 | 70%    | Essay (25%) and large programming task (45%) |

## Some words about lectures, marking, auto-testing and essays

As software engineering changes, so does ILP. In the last years ILP was mainly a coding course, where you got some specific instructions (this document), implemented some classes and got some result. This no longer serves the purpose; You should be able to reflect what you are doing, and implement the features necessary, including unit-test.

To address these issues, several changes were made to the ILP **lecture structure**, which are outlined below:

- As REST-services and containerization are now commonplace, ILP will be implemented as a REST-service running inside a docker container as well.
- To enable you doing that and create enough background knowledge, many lectures about these topics will be provided. This will include JSON-handling as well as REST-server-access.
- Several programming specific lectures will cover more advanced topics like i.e. object oriented design issues, new Java features like streams, lambda functions, etc.;

**Marking** is always a difficult topic in such large a course and especially to have the right balance between pure coding marks and marks for your insights (in terms of essays). Many of you (usually the ones who are rather enjoying development), are often not too keen on essay writing and vice versa. Yet the balance is important, now and in a later job as well. Very often you will hit a wall where you have to argue for something - a change, a new feature, a pay-rise, whatever. These arguments and reasoning will be in written form as well, and then it pays out to have a sound basis for that.

Therefore, we are trying to balance marking a bit by reserving 25% of the final mark for the essay where you can explain your decisions, choices, considerations and thoughts. As you can see, **auto-marking** will be a large part of your final mark. The benefit of auto-marking is that there is no unconscious bias, no personal preferences of style or form - just rules which have to be fulfilled. So we tried extremely hard (learning from the previous courses as well) to have all this as smooth as possible for you.

To get you used to auto-marking, coursework 1 will only consist of auto-marking (with a very moderate impact on the final mark) with a limited requirement specification, to help you pass as good as possible.

## Specifications for CW1 and CW2 (programming task and essay)

As CW1 and CW2 change more often than this more general document, **we decided to separate the detailed specifications for CW1 and CW2.**

So, for the detailed spec, please access the corresponding Learn page in ILP (usually *Assessment Instructions* under *Assessment*) for any specific information.

# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>The Coursework Specification</b>   | <b>5</b>  |
| 1.1      | Introduction . . . . .  | 5         |
| 1.2      | Latitudes and longitudes . . . . .  | 7         |
| 1.3      | Drone movement specifications and definitions . . . . .                             | 8         |
| 1.4      | The University of Edinburgh Central Area . . . . .                                  | 10        |
| 1.5      | The ILP-REST-Service . . . . .  | 10        |
|          | REST-Service Endpoints for data . . . . .   | 13        |
|          | JSON-data wrapping (using de/serialization) . . . . .                               | 13        |
| 1.6      | JSON data objects . . . . .   | 13        |
| 1.6.1    | The makeup of an order . . . . .  | 15        |
| 1.6.2    | normal flow for order processing . . . . .  | 16        |
| 1.6.3    | Order limitations . . . . .   | 16        |
| 1.6.4    | Order validation . . . . .  | 17        |
| 1.7      | Your challenges during the implementation . . . . .                                 | 19        |
| 1.7.1    | Additional files to be used during testing . . . . .                                | 19        |
| 1.7.2    | An illegal flight path . . . . .  | 20        |
| 1.8      | Programming language: Java . . . . .  | 21        |
| <b>2</b> | <b>Informatics Large Practical: Considerations for Programming Task Submissions</b> | <b>22</b> |
| 2.1      | What has to be submitted . . . . .  | 22        |
| 2.2      | Source code of your service . . . . .   | 22        |
| 2.3      | Unit testing your service . . . . .   | 23        |
| 2.4      | No files are created in assignments . . . . .                                       | 23        |
| 2.5      | Things to consider . . . . .  | 24        |
| 2.6      | How to submit . . . . .   | 24        |
| <b>A</b> | <b>Coursework Regulations</b>   | <b>25</b> |
| A.1      | Good scholarly practice . . . . .   | 25        |
| A.2      | Late submission policy . . . . .  | 25        |

# List of Figures

|     |   |    |
|-----|---|----|
| 1.1 | The 16-point compass rose. Original SVG image <a href="https://commons.wikimedia.org/w/index.php?curid=2249878">https://commons.wikimedia.org/w/index.php?curid=2249878</a> . . . . .   | 9  |
| 1.2 | The University of Edinburgh Central Area . . . . .  | 10 |
| 1.3 | Showing the start page of the ILP REST Service . . . . .  | 11 |
| 1.4 | Showing the central area data for the REST-request for <i>centralArea</i> . . . . .   | 12 |
| 1.5 | Showing an error as an access was attempted with a resource being specified which results in a HTTP code <b>404</b> - Resource not found . . . . .  | 12 |
| 1.6 | The contents of the file <b>all.geojson</b> rendered by the website <a href="http://geojson.io/">http://geojson.io/</a> . This file contains all of the features that we have seen in Figure ?? plus the initial location of the drone (the yellow placemaker, on top of Appleton Tower), and the four pizza restaurants which are participating in the scheme according to the website content <b>restaurants.geojson</b> (the blue placemarkers, with a building symbol). The semi-transparent red polygons are the no-fly zones. . . . . | 20 |
| 1.7 | An illegal flightpath which leaves the Central Area again after having entered it. . . .  | 20 |
| 2.1 | A previous year's example rendering of the base map ( <b>all.geojson</b> ) overlaid with several drone flightpaths (manually concatenated). The rendering was done by the website <a href="http://geojson.io/">http://geojson.io/</a> . Note that the drone never enters the no-fly zones (the semi-transparent red polygons). . . . .  | 23 |

# Chapter 1

## The Coursework Specification

### 1.1 Introduction

Have you ever been involved in an all-night hackathon or have been pulling an all-nighter to get super-tough practicals like this one finished on time? If so you will know that there comes a time late at night when nothing else but a pizza will keep you going to get all your work done! Well, not to worry, the School of Informatics is considering creating a service called *PizzaDronz* where students can order a pizza by an app and have it delivered directly by drone to the top of the Appleton Tower where they can collect it and eat it while taking a break from the keyboard. Your midnight feast worries should be a thing of the past when the service launches on 1st January 2025! Of course, pizzas make a great lunchtime snack to share with friends so the service will operate all day, not just during the hours of darkness.

— ◇ —

Having pizzas delivered by drone will minimise the time that busy Informatics students need to spend queuing to buy lunch or dinner and will also speed delivery, because, unlike delivery by car or bike, drones do not need to follow the road layout, stop at red traffic lights, and so forth. The idea is also good for the environment. Fewer deliveries by car means less exhaust pollution generated, and cleaner air for Edinburgh. In addition, the service could be helpful to new students who have just joined the School and have not yet got their bearings and do not know the great pizza restaurants near the University's Central Area.

— ◇ —

The idea for a drone-based food delivery service has some issues, especially when delivering hot food. We will assume that the drone will fly high enough that it will not crash into even very tall buildings such as The David Hume Tower. However, Edinburgh's many seagulls might think that someone has kindly sent them a delicious meal and attack the drone in order to be able to get at the contents inside. In addition, software and hardware errors do happen so we must route the drone as much as possible away from populated areas such as George Square Gardens and Bristo Square, among others. Ideally, the drone should usually be flying over the roofs of buildings. This is for the *safety* reason just mentioned (we don't want to drop hot food or metal drones onto unsuspecting students studying in the sunshine in George Square Gardens). An additional issue is *privacy*. Many drones are fitted with cameras and some students might not like the idea that they might be photographed by a drone flying overhead. If the drone is flying over the roofs of buildings then a more innocent explanation might occur such as the University is surveying the roofs of the buildings and looking for cracked roof tiles or leaks in a roof. In fact, the drone does not have a camera fitted but privacy is important and we don't want to give students unnecessary worries about their privacy. For this reason, student-populated

areas will be designated as “no-fly zones”. The drone will therefore have to plan its routes so that it does not fly over the no-fly zones.

— ◇ —

Ordering the pizzas is relatively, but not completely straightforward. The School of Informatics is developing an online system to take pizza orders and add these to a database of orders to be delivered. This online system will consist of 4 main parts:

- ▷ an order processing back-end which is currently under development <sup>1</sup>.
- ▷ a REST-based service, which serves more static information like restaurants, no-fly-zones, opening hours, menus, static files (like maps), etc.  
You will be consuming the service (in part 2 of the course), yet not modify anything. The base URL is: `https://ilp-rest-2024.azurewebsites.net/`
- ▷ **A REST-based service to do "the heavy lifting" by calculating the routes for the drone from restaurant to AT, to check orders and simple flight calculations.** <sup>2</sup>
- ▷ A mobile user-interface probably in react-native to cover iOS and android at the same time

The main problem is that when an order is placed in the UI it is not clear whether or not it is feasible for the drone to fulfil these orders, given that (i) the service is expected to be popular with a lot of pizza orders being placed each day, (ii) only one drone is available for making the deliveries, (iii) the drone cannot carry more than one order (of a maximum of four pizzas) at a time (to avoid delivering the wrong order to the wrong person, for example a non-vegetarian pizza to a vegetarian), (iv) the drone must avoid populated areas in the no-fly zone, (v), and specifically, (vi) the drone can carry between one and four pizzas<sup>3</sup>. In addition, several participating restaurants have different opening days, so not every choice is available every day.

— ◇ —

Your task is to devise and implement a REST-based micro-service which provides endpoints to:

- ▷ Get the drone’s flight-path from restaurant to AT as coordinate set
- ▷ Get the drone’s flight-path from restaurant to AT as GetJSON set
- ▷ Validate orders
- ▷ Perform flight path calculations <sup>4</sup>

— ◇ —

To make the services as available and scalable (vertical vs horizontal scaling) as possible, they will be run in a kubernetes environment later. For the development time docker will be utilized as this is a similar container environment with less effort and complexity. Thus, your service will have to run as a docker container inside docker.

A benefit of that is that your service can run completely on its own in its own runtime environment, not affected by anything on the outside. It just provides "endpoints" (like /isalive, etc) which can be called and return data according to the requirements (coursework specifications).

---

<sup>1</sup>and not to be considered further

<sup>2</sup>This is your work and the main subject of the ILP course

<sup>3</sup>It is not possible to order zero or negative numbers of pizzas, or more than four pizzas, half-pizzas, single slices of pizza, or drinks or ice-cream or any other types of snacks. Pizzas are only available in a maximum of 14-inch size. It is not possible to order larger pizzas than 14-inch pizzas, but smaller diameter ones are OK.

<sup>4</sup>which acts more as a "warm-up" in phase 1 for the challenges of phase 2

— ◇ —

You will be provided with data about such details as the pizza restaurants which are participating in the scheme, the menus for these shops, and the location of the drop-off point on top of the Appleton Tower. This information will come in the form of a REST-service running on a server, which returns the data in JSON-format<sup>5</sup> (*more detailed information and an example will be provided*).

It is important to stress that the information in the test data which you will be given only represents the current best guess at what the elements of the drone service will be when it is operational, and the service in practice might use different shops or it might even deliver to different drop-off points (such as the top of the Informatics Forum). For this reason, your solution must be *data-driven*. That is, it must read the information from the REST-service and particular shops or particular drop-off points or other details must not be hard-coded in your application, except where it is explicitly stated in this document that it is acceptable to do so.

— ◇ —

As we are currently in a start-up and experimental phase, the data is not in optimal shape and many orders are invalid due to various reasons.

Among those (*the full list of error codes is in an enumeration inside `IlpDataObjects.jar`*) are: invalid card numbers, expiration dates, pizzas in invalid combinations, orders for restaurants which are closed, wrong size pizza, etc.

— ◇ —

The *PizzaDronz* operators have a set of thermally-insulated boxes which are attached to the drones. When a full insulated box of pizzas lands on the top of the Appleton Tower it is swapped for an empty insulated box and sent off for its next delivery run. The insulated boxes are thoroughly sanitised between uses.

— ◇ —

You should think that your software is being created with the intention of passing it on to a team of software developers and student volunteers in the School of Informatics who will maintain and develop it in the months and years ahead when the *PizzaDronz* delivery service is operational. For this reason, the *clarity and readability of your code is important*; you need to produce code which can be read and understood by others.

— ◇ —

AS stability is of paramount importance for such a service, your code will have to have suitable unit-testing (for the REST-interface as well as the internal logic).

## 1.2 Latitudes and longitudes

In this practical we will be using latitudes and longitudes to identify locations on the map (such as pizza restaurants and the drop-off point on the roof of the Appleton Tower).

- ▷ Longitude is the measurement east or west of the prime meridian.
- ▷ Latitude is the measurement of distance north or south of the Equator.

---

<sup>5</sup><https://en.wikipedia.org/wiki/JSON>

(The above are National Geographic definitions.) Latitudes and longitudes are measured in *degrees*, so we stay with this unit of measurement throughout all our calculations. Even when we are calculating the *distance* between two points we express this in degrees rather than metres or kilometres to avoid unnecessary conversions between one unit of measurement and another.

As a convenient simplification in this practical, locations expressed using latitude and longitude are treated as though they were points on a plane, not points on the surface of a sphere. This simplification allows us to use Pythagorean distance as the measure of the distance between points. That is, the distance between  $(x_1, y_1)$  and  $(x_2, y_2)$  is just

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

—  $\diamond$  —

In general, it will not be possible to manoeuvre the drone to a specified location exactly. Being *close* to the location will be sufficient, where  $\ell_1$  is *close* to  $\ell_2$  if the distance between  $\ell_1$  and  $\ell_2$  is strictly less than the *distance tolerance* of 0.00015 degrees.

—  $\diamond$  —

When we write a location as a pair of coordinates in this document we will use the convention (*longitude*, *latitude*) because the language which we use for rendering maps puts longitude first and latitude second. In this project, longitudes will always be negative ( $\sim -3$ ) and latitudes will always be positive ( $\sim +56$ ) because Edinburgh is located at (approximately) longitude 3 degrees West and latitude 56 degrees North.

### 1.3 Drone movement specifications and definitions

The flight of the drone and its movement is subject to the following stipulations:

- the moves are of two types, the drone can either *fly* or *hover*—the drone can change its latitude and longitude when it flies, but not when it is hovering i.e. when it makes a hover move—flying and hovering use the same amount of energy;
- every move when flying is a straight line of length 0.00015 degrees<sup>6</sup>;
- the drone *cannot fly in an arbitrary direction*: it can only fly in one of the 16 major compass directions as seen in Figure 1.1. These are the primary directions North, South, East and West, and the secondary directions between those of North East, North West, South East and South West, and the tertiary directions between those of North North East, East North East, and so forth. We use the convention that 0 means go East, 90 means go North, 180 means go West, and 270 means go South, with the secondary and tertiary compass directions representing the obvious directions between these four major compass directions. The convention that we use for angles simplifies the calculation of the next position of the drone.
- as the drone flies, it travels at a constant speed and consumes power at a constant rate;
- when the drone is hovering, we use **999**<sup>7</sup> as the reference value for the angle, to indicate that the angle does not play a role in determining the next latitude and longitude of the drone;

---

<sup>6</sup>Because of unavoidable rounding errors in calculations with double-precision numbers these moves may be fractionally more or less than 0.00015 degrees. Differences of  $\pm 10^{-12}$  degrees are acceptable. Double-precision numbers must be used to represent quantities measured in degrees because of the need for accuracy in specifying locations.

<sup>7</sup>Using null would require Float instead of float as datatype and in JSON this would cause other problems as well. So we opted to use 999 as distinguishing value



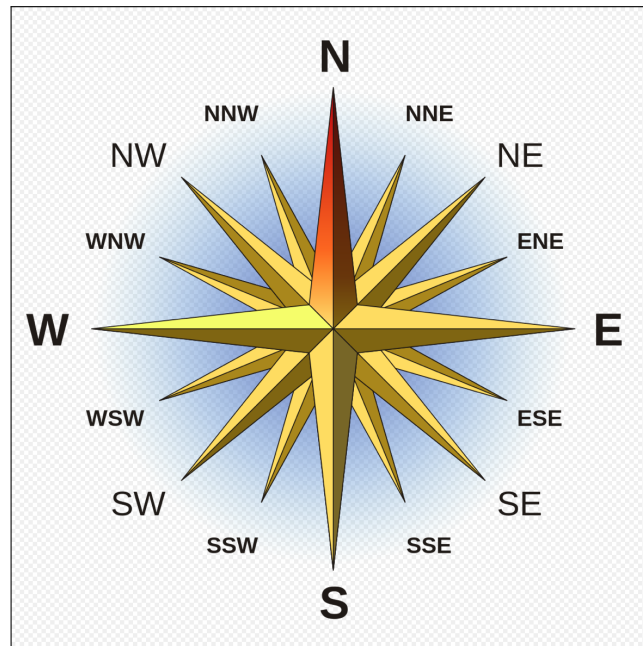


Figure 1.1: The 16-point compass rose. Original SVG image <https://commons.wikimedia.org/w/index.php?curid=2249878>

- the drone *must hover for one move* when collecting a pizza order from a restaurant, and do the same when delivering pizzas to the roof of the Appleton Tower;
- if the drone hovers start and end coordinates of the move are identical (as it hovers in mid-air, not moving at all)
- as the trip from AT to the restaurant is the same as the return way, you only have to calculate the routes from the restaurants to the top of the Appleton Tower at location  $(-3.186874, 55.944494)$

## 1.4 The University of Edinburgh Central Area

The University of Edinburgh Central Area is defined to be all locations which have a latitude which lies between 55.942617 and 55.946233. They also have a longitude which lies between  $-3.184319$  and  $-3.192473$ . Outside organisations should have little objection to the drone being in this area because this is mostly University land containing University buildings. For this reason it is important for the drone to return to the Central Area once it has collected the pizza(s) *in as few moves as possible*.

The Central Area is illustrated in Figure 1.2.

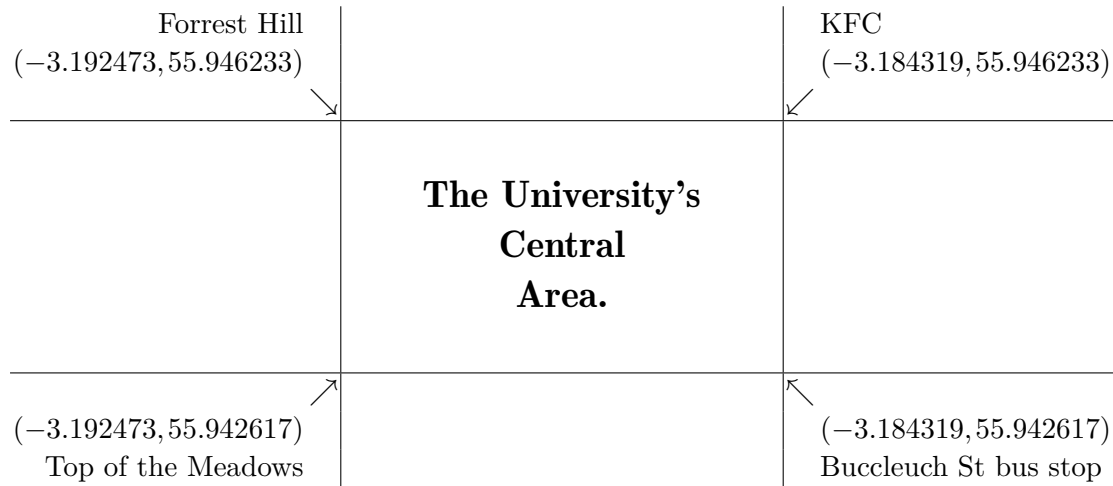


Figure 1.2: The University of Edinburgh Central Area

**The above constants are merely used for representational purpose.** As we try to be as much data-driven as possible, the central area must be queried from the ILP-service via <https://ilp-rest-2024.azurewebsites.net/centralArea> which will return a **named region** which defines a **closed** polygon with vertices as the corner points. **Closed** as the last point will refer back to the start.

Once the drone has entered the Central Area, *it cannot leave it again* until it has delivered the ordered pizzas to the roof of the Appleton Tower. This rule prevents invalid solutions which attempt to avoid the no-fly zones by leaving the Central Area and returning into it just beside Appleton Tower.

## 1.5 The ILP-REST-Service

All dynamic information which the PizzaDronz service needs is provided by a newly developed centralized REST-service (base URL at <https://ilp-rest-2024.azurewebsites.net/>). This service (which actually could be a single machine or a cluster to provide a more secure platform for later business growth) makes the whole system more dynamic and your life much easier as it serves as a central point of intelligence.

The REST-service provides a REST-API for dynamic data, which allows you to retrieve always up-to-date information about orders, restaurants, zones, etc.

**All data needed by the *PizzaDronz* service is coming from the ILP-REST-service and has to be retrieved every time the *PizzaDronz* service is starting (which could be for every request) to make sure the latest information is processed.** This is very important as otherwise data changes might not be reflected properly.

To test the REST-server you can use your browser and just type the base-URL for the base-page or the REST-endpoint (for dynamic resources).

The Base-URL will produce the following image:

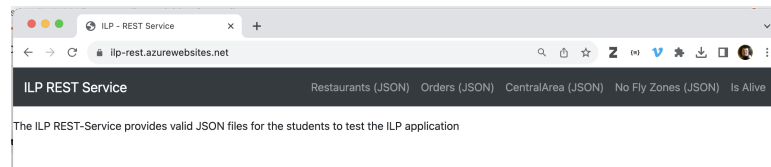
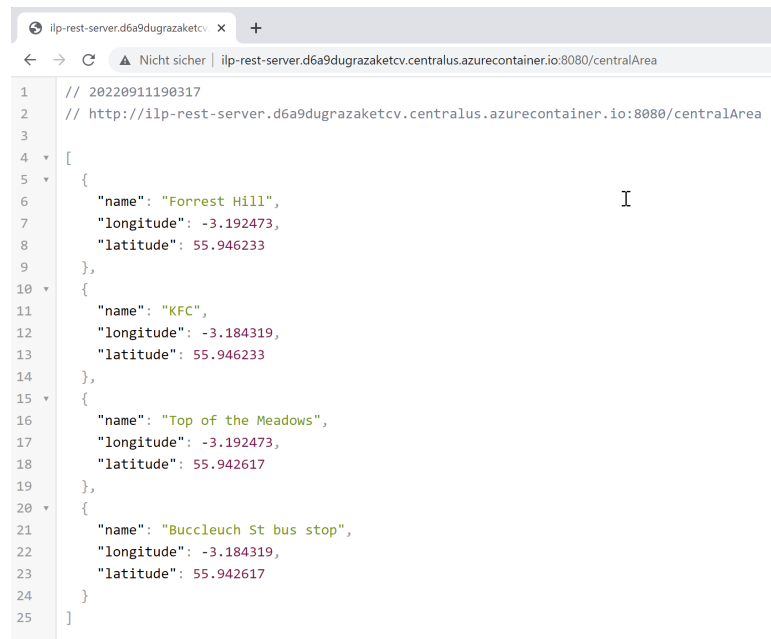


Figure 1.3: Showing the start page of the ILP REST Service



The result of the operation is always rendered in your browser; either as html (for the base page) or the display of a JSON data structure (the dynamic data). This could look as in Figure 1.4 (for the central area data):



```

1 // 20220911190317
2 // http://ilp-rest-server.d6a9dugrazaketcv.centralus.azurecontainer.io:8080/centralArea
3
4 [
5   {
6     "name": "Forrest Hill",
7     "longitude": -3.192473,
8     "latitude": 55.946233
9   },
10  {
11    "name": "KFC",
12    "longitude": -3.184319,
13    "latitude": 55.946233
14  },
15  {
16    "name": "Top of the Meadows",
17    "longitude": -3.192473,
18    "latitude": 55.942617
19  },
20  {
21    "name": "Buccleuch St bus stop",
22    "longitude": -3.184319,
23    "latitude": 55.942617
24  }
25 ]

```

Figure 1.4: Showing the central area data for the REST-request for *centralArea*

— ◇ —

Should an error occur (either by accessing an invalid resource *URL* or a server-side problem) an error display similar to Figure 1.5 is shown



Figure 1.5: Showing an error as an access was attempted with a resource being specified which results in a HTTP code **404** - Resource not found

— ◇ —

These URLs could look like:

- <https://ilp-rest-2024.azurewebsites.net/><sup>8</sup>
- <https://ilp-rest-2024.azurewebsites.net/restaurants>

---

<sup>8</sup>the base page

- <https://ilp-rest-2024.azurewebsites.net/noFlyZones>
- <https://ilp-rest-2024.azurewebsites.net/centralArea>
- <https://ilp-rest-2024.azurewebsites.net/bounding-box.geojson>
- <https://ilp-rest-2024.azurewebsites.net/all.geojson>

— ◇ —

### REST-Service Endpoints for data

- **centralArea** - returns the central area corner points (currently 4) starting top left and then in anti-clockwise direction defining a rectangle.
- **noFlyZones** - lists all defined no-fly zones as an array of objects with a name and polygon coordinates
- **restaurants** - lists all defined restaurants with their coordinates and the pizzas on offer (including a price for each pizza)

### JSON-data wrapping (using de/serialization)

The data returned is just a sequence of characters and by using *Jackson*<sup>9</sup> or *Gson*<sup>10</sup> this is converted to a class instance<sup>11</sup>.

## 1.6 JSON data objects

Inside the ILP- and PizzaDronz-service the following JSON objects are being used:

- **LngLat** which defines a point with "lng" and "lat" as longitude and latitude. Typical JSON would look like

```
1 {
2   "lng": -3.192473,
3   "lat": 55.946233
4 }
```

- **NamedRegion** defines an array of points (vertices) which form a closed<sup>12</sup> polygon. To distinguish each named region can have a name.

```
1 {
2   "name": "central",
3   "vertices": [
4     {
5       "lng": -3.192473,
6       "lat": 55.946233
7     },
```

<sup>9</sup>For detailed information please visit: <https://stackabuse.com/definitive-guide-to-jackson-objectmapper-serialize-and-deserialize-java-objects/>

<sup>10</sup><https://github.com/google/gson/>

<sup>11</sup>This process of converting the textual data to objects is called de-serialization

<sup>12</sup>as the last point will lead back to the start

```

8      {
9          "lng": -3.192473,
10         "lat": 55.942617
11     },
12     {
13         "lng": -3.184319,
14         "lat": 55.942617
15     },
16     {
17         "lng": -3.184319,
18         "lat": 55.946233
19     },
20     {
21         "lng": -3.192473,
22         "lat": 55.946233
23     }
24 ]
25 }

```

- **Pizza** defines a class with a name and a price which is an integer

```

1 {
2     "name": "Margarita",
3     "priceInPence": 1000
4 }

```

- **Restaurant** defines a location with a name, opening days and a menu of available pizzas. The days are derived from the Java class **DayOfWeek**. Restaurants can be open on different days and thus pizzas are not always available.

```

1 {
2     "name": "Civerinos Slice",
3     "location": {
4         "lng": -3.1912869215011597,
5         "lat": 55.945535152517735
6     },
7     "openingDays": [
8         "MONDAY",
9         "TUESDAY",
10        "FRIDAY",
11        "SATURDAY",
12        "SUNDAY"
13    ],
14    "menu": [
15        {
16            "name": "Margarita",
17            "priceInPence": 1000
18        },
19        {
20            "name": "Calzone",

```

```

21     "priceInPence": 1400
22   }
23 ]
24 }

```

- **OrderValidationResult** defines the result after validating an order and contains a status for the whole order and a further validation code if an error occurred:

```

1 {
2   "orderStatus": "VALID",
3   "orderValidationCode": "NO_ERROR"
4 }

```

- **CreditCardInformation** defines the associated data with credit cards. In ILP and PizzaDronz we are only using 16 digit numbers and 3 digit CVVs.

```

1 {
2   "creditCardNumber": "4485959141852684",
3   "creditCardExpiry": "10/25",
4   "cvv": "816"
5 }

```

- **Order** as the most complex type is defined below in more detail as an order contains a lot of semantic conditions.

Necessary derived JSON-types can be easily constructed. For details of the actual data types to use, please check the result of the ILP-service endpoints or the corresponding CW-specification.

### 1.6.1 The makeup of an order

An order looks like the following JSON example:

```

1 {
2   "orderNo": "092884D0",
3   "orderDate": "2024-01-28",
4   "priceTotalInPence": 2400,
5   "pizzasInOrder": [
6     {
7       "name": "R6: Sucuk delight",
8       "priceInPence": 1400
9     },
10    {
11      "name": "R6: Dreams of Syria",
12      "priceInPence": 900
13    }
14  ],
15   "creditCardInformation": {
16     "creditCardNumber": "4485959141852684",
17     "creditCardExpiry": "10/25",
18     "cvv": "816"

```

```

19   }
20   }

```

It contains:

- **orderNo** as unique order id
- **orderDate** as date which for validation must be  $\geq$  now
- **priceTotalInPence** as the sum of all pizzas in the order
- **pizzasInOrder** as an array of pizzas
- **creditCardInformation** as the card info to be used for payment

*Always keep in mind that the JSON you receive might have more data than in the definition before, which you will simply ignore and proceed with the relevant data items.*

**This data structure is not ideal and will be changed in the future. Yet for the time being we have to use it and apply additional validation to prevent e.g. pizzas being ordered from different restaurants, etc**

*Every order you want to process has to be validated and if not valid no processing can happen. In such cases you will return a corresponding return code from the HTTP request like 400*

### 1.6.2 normal flow for order processing

The normal flow an order would be processed is:

- An order is sent to the PizzaDronz service for validation and a validation result is processed
- After the validation is done and the order is **VALID** the route can be retrieved. The routine has to check the validity as well to make sure only valid orders are processed
- If a route request is retrieved for an invalid order **Bad Request** should be returned

### 1.6.3 Order limitations

We haven't said much so far about the nature of a pizza order so let's discuss that now. As you might imagine, there is a limit on the weight that the drone can lift. The maximum number of items in an order has been fixed so that the drone will always be able to lift the order, even if it consists of the heaviest pizzas which can be ordered by the drone service. There are other constraints also, as listed below.

1. An order can have a minimum of one pizza, and a maximum of four.
2. Every order is subject to a fixed delivery charge, which is £1.

The restaurants which participate in the service are notified when the drone will arrive; they start cooking the pizza(s) and then when the drone arrives they place the pizzas in an insulated box, and fix the box to the drone when it is hovering close to the location of the restaurant/pizza shop. We imagine that the insulated box is hanging down from the drone so that the drone is always hovering some safe height above the user's head.



The box can contain pizzas up to 14 inches in diameter, but not larger than this. None of the pizzas returned by the REST-request to `restaurants` are larger than this.

— ◇ —

We will not be very concerned here with the system which sends web or text message order notifications to the shops; the architects of the drone service already have a system in place for this. However, due to a misunderstanding between the web front end back-end team, each thought that the other was responsible for validating the data entered by the customer and as a result neither team has implemented this. This means that there will be invalid orders in the REST-responses from `orders` which you must detect and filter out and not attempt to deliver. We will give examples later.

#### 1.6.4 Order validation

Validation of an order yields a validation code and an order status. The listings below are derived from the corresponding Java files in the `IlpDataObjects` repository on GitHub.

The following validation codes are defined:

```
1 public enum OrderValidationCode {
2     /**
3      * the reason code is undefined
4      */
5     UNDEFINED,
6
7     /**
8      * no error present
9      */
10    NO_ERROR,
11
12    /**
13     * the card number is incorrect
14     */
15    CARD_NUMBER_INVALID,
16
17    /**
18     * expiry date problem
19     */
20    EXPIRY_DATE_INVALID,
21
22    /**
23     * CVC is wrong
24     */
25    CVV_INVALID,
26
27    /**
28     * order total is incorrect
29     */
30    TOTAL_INCORRECT,
31
32    /**
33     * a pizza in the order is undefined
```

```

34     */
35     PIZZA_NOT_DEFINED,
36
37     /**
38      * too many pizzas ordered
39      */
40     MAX_PIZZA_COUNT_EXCEEDED,
41
42     /**
43      * pizzas were ordered from multiple restaurants
44      */
45     PIZZA_FROM_MULTIPLE_RESTAURANTS,
46
47     /**
48      * the restaurant is closed on the order day
49      */
50     RESTAURANT_CLOSED,
51
52     /**
53      * a pizza was ordered with an invalid price
54      */
55     PRICE_FOR_PIZZA_INVALID,
56
57     /**
58      * the order contains no pizzas
59      */
60     EMPTY_ORDER
61 }

```

The following order status is defined:

```

1  /**
2   * the status an order can have
3   */
4  public enum OrderStatus {
5
6      /**
7       * it is invalid
8       */
9      INVALID,
10
11     /**
12      * the state is valid
13      */
14     VALID,
15
16     /**
17      * the state is currently undefined
18      */
19     UNDEFINED

```

20 }

## 1.7 Your challenges during the implementation

Your PizzaDronz service has to provide the following functionality:

- Validate orders
- Calculate flight-paths and return them
- Calculate GetJSON representations for a flight-path and return them

A flight-path of the drone for an order should aim to have a runtime of *60 seconds or less* considering a rather powerful hosting machine for the docker runtime (where a good implementation would yield around 5-10 sec). You need to bear this restricted runtime in mind when designing the algorithm that you will use to generate the flight-path of the drone.

— ◇ —

Runtimes which are much longer than 60 seconds, would obviously be problematical because they are delaying the launch of the drone considerably. Delays in the collection and delivery of their pizza(s) are sure to be unpopular both with the restaurants and with hungry students so an algorithm with a runtime of more than the above stated limit would be quite unsatisfactory and result in less points achieved.

— ◇ —

Saying this, please do not focus too much on the performance of the algorithm, as long as it stays somehow within reasonable dimensions. The overall architecture and result data structures, as well as order retrieval, processing and cleansing are equally important and should require your attention as well.

— ◇ —

### 1.7.1 Additional files to be used during testing

In addition to the dynamic data returned from the various JSON requests while your application is running, you can retrieve some GeoJSON files<sup>13</sup> which have been prepared for you to use when you are *testing* your application.

— ◇ —

When the *PizzaDronz* service is operational the server content will be kept up-to-date but the GeoJSON files prepared for testing purposes will not; they only relate to the synthetic data that you are given to test your application, not the lunch orders received each day when the service is operational. This means that your application should not read these GeoJSON files, you should only load them on the <http://geojson.io/> website when checking the flightpath that you have generated for the airborne drone.

— ◇ —

<sup>13</sup>see above for details or just download the files using the browser and keep them locally

Unlike, for example, the GeoJSON file of the no-fly zones, the content of the database is not held in a graphical format so the purpose of the GeoJSON test files is to allow us to produce a visualisation of the information in the database, to help with understanding the important locations which are in the synthetic data. One file, `all.geojson`, includes representations of all of the locations of interest in the drone Central area. This is shown in Figure 1.6. This file will make a convenient background when rendering your drone flightpath.

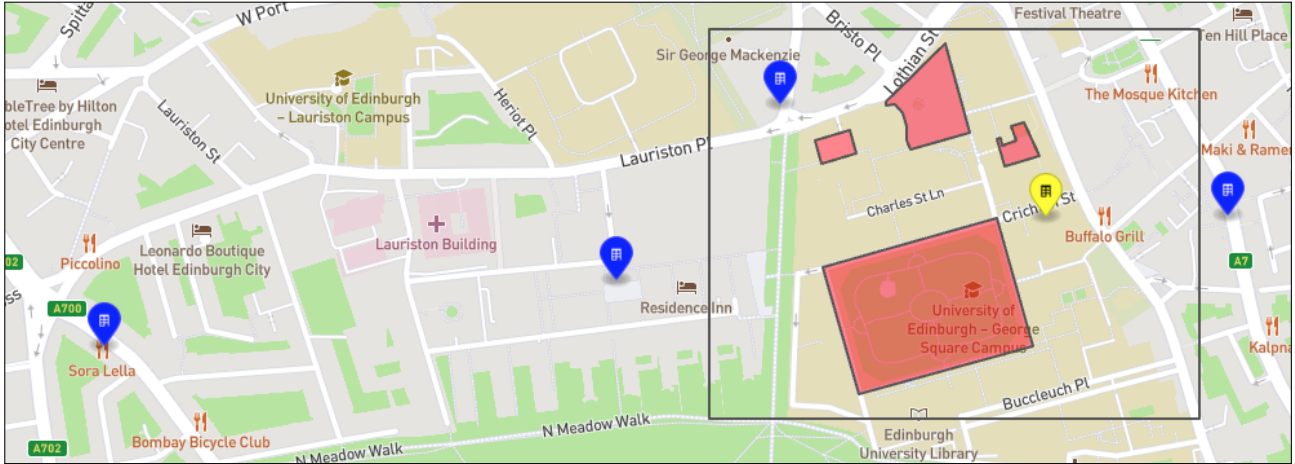


Figure 1.6: The contents of the file `all.geojson` rendered by the website <http://geojson.io/>. This file contains all of the features that we have seen in Figure ?? plus the initial location of the drone (the yellow placemaker, on top of Appleton Tower), and the four pizza restaurants which are participating in the scheme according to the website content `restaurants.geojson` (the blue placemarkers, with a building symbol). The semi-transparent red polygons are the no-fly zones.

### 1.7.2 An illegal flight path

Once back inside the Central Area, the drone must not leave again until it has delivered the pizzas to the roof of Appleton Tower. The flightpath shown in Figure 1.7 from Sora Lella restaurant to Appleton Tower is completely illegal, as well as being sub-optimal in taking more moves than necessary.

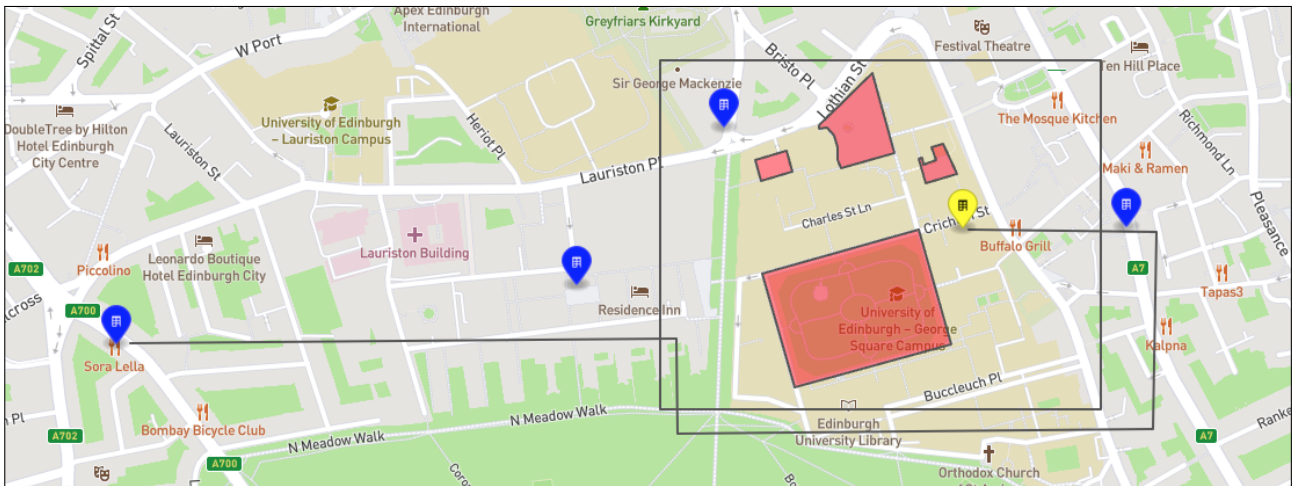


Figure 1.7: An illegal flightpath which leaves the Central Area again after having entered it.

## 1.8 Programming language: Java

The programming language to be used for your service is Java and the whole (including all libraries, etc) must run inside a docker runtime as a container in an AMD64 environment.

— ♦ —

You are free to use any needed libraries and Java-versions as long as you can produce a valid docker image

— ♦ —

We expect you to be already familiar with Java. If you are not, or if you would benefit from a refresher on Java, we recommend the textbook *Java Precisely* by Peter Sestoft, third edition published by MIT Press in 2016, as providing a concise and clear introduction to Java.<sup>14</sup>

---

<sup>14</sup>Many Java textbooks are available so if you cannot get a copy of *Java Precisely* please feel free to choose another textbook. Alternatively, many tutorials on Java are available online, including the (slightly dated but still very useful) Java Tutorial from Oracle at <https://docs.oracle.com/javase/tutorial/>. The Java real-eval-print loop `jshell` can also provide a useful refresher.

## Chapter 2

# Informatics Large Practical: Considerations for Programming Task Submissions

---

Michael Glienecke

School of Informatics, University of Edinburgh

Document version 4.1.0

**This chapter only provides general information. For details, please refer to the ILP Learn page for the CW1 and CW2 programming task requirements**

---

### 2.1 What has to be submitted

Any programming task submission consists of a ZIP file of the entire working directory structure of your project, as well as the exported docker image you created as a tar-file in the root directory.

From this the auto-marker will extract the tar-file and import it into a docker container (therefore the AMD64 is important to make sure the import works). The sources are necessary proof and evidence of your independent work, as well as should something fail with the tar-file.

**If no sources or no tar-file are present, marking cannot happen. This will result in 0 as assignment result.**

### 2.2 Source code of your service

You are submitting your source code for review where your Java code will be read by a person, not a script. You should tidy up and clean up your code before submission. This is the time to remove commented-out blocks of code, tighten up visibility modifiers (e.g. turn `public` into `private` where possible), fix and remove TODOs, rename variables which have cryptic identifiers, remove unused methods or unused class fields, fix the static analysis problems which generate warnings from IntelliJ<sup>1</sup>, and refactor your code as necessary. The code which you submit for assessment should be well-structured, readable and clear.

---

<sup>1</sup>normally there has to be a very good reason why any warning should still be issued after this clean-up phase

## 2.3 Unit testing your service

Your service needs to be tested properly and unit-tests (covered in the lectures) are a major part of that and considered the "internal" test (compared to any outside test using Postman, etc). Therefore, for CW2 the quality of your provided tests will be marked as well (although only with a small percentage of the total mark).

## 2.4 No files are created in assignments

The service only returns JSON-data and / or http status codes from the exposed API. Therefore, no direct files are returned, yet can be generated by saving the response into a corresponding file.

Especially the GeoJSON file-format (for the corresponding endpoints) can be interesting as with that the base map <sup>2</sup> could be combined with the response of a drone-route-request to produce an image like:

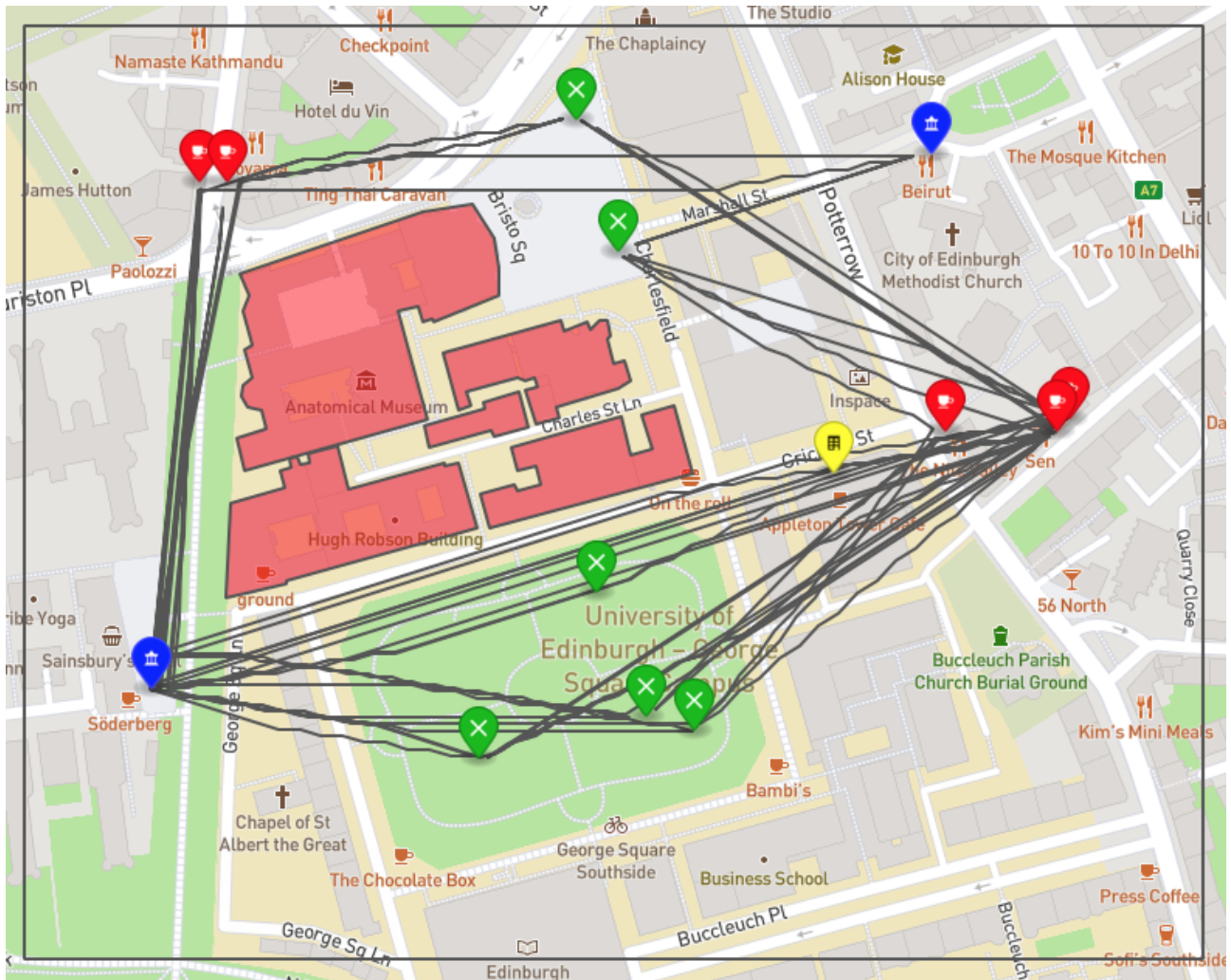


Figure 2.1: A previous year's example rendering of the base map (`all.geojson`) overlaid with several drone flightpaths (manually concatenated). The rendering was done by the website <http://geojson.io/>. Note that the drone never enters the no-fly zones (the semi-transparent red polygons).

<sup>2</sup>retrievable from the ILP-web-service, normally at <https://ilp-rest-2024.azurewebsites.net/all.geojson>

## 2.5 Things to consider

- Please make sure that your submitted tar-file really runs properly. This can be easily tested by simply importing it into your local docker system and running it plus accessing with postman or curl
- Your code will be executed "as is" inside the target docker environment (with 8080 as assumed local port) via the provided tar-image.  
Therefore, it can be assured that anything you implemented for your service in your development system is 100% identically executed on the target system.
- Should you produce initialization exceptions (e.g. Java version wrong, etc) then the service will not work / run and cannot be tested (resulting in a 0 mark).
- Your submitted Java code will be read and assessed by a person, not a script. It should contain helpful comments in JavaDoc format, documenting your intentions. Your submitted code should be readable and clear.
- Logging statements and diagnostic print statements (using `System.out.println` and friends) are useful debugging tools. You do not need to remove them from your submitted code; it is fine for these to appear in your submission. You can write whatever you find helpful to `System.out`.  
An excessive level of logging can be counter-productive, causing the user not to read the log output, thereby defeating the purpose. Consider what should be logged, and log sparingly.
- Error messages should be written to `System.err`, not `System.out`.
- Your service should be robust and always only return the defined http codes regardless of the call issued. Failing with any kind of exception or other run-time error inside the docker environment will be considered a serious fault.

## 2.6 How to submit

Ensure that you are LEARN-authenticated by visiting <http://learn.ed.ac.uk>. Go to the ILP LEARN page. Click on the *Assessment* link in the left-hand margin bar and then the corresponding link for the CW1 or CW2 programming task. Use the *Browse Local Files* option to find and upload your files

- `sXXXXXXX.zip` - here XXXXXXXX is your student id

The archive format for compressed files is ZIP only; **do not submit TAR, TGZ, or RAR files, or other formats**. When finished, make sure that you click *Submit*.

— ◇ —

This submission mechanism should allow you to make multiple submissions. Later submissions will overwrite earlier ones and the last (and only the last!) submission will be marked.

Submissions which arrive after the coursework deadline will be subject to the School's late submission penalties as detailed at <http://web.inf.ed.ac.uk/infweb/student-services/ito/admin/coursework-projects/late-coursework-extension-requests>.



# Appendix A

## Coursework Regulations

---

### A.1 Good scholarly practice

Please remember the good scholarly practice requirements of the University regarding work for credit. You can find guidance at the School page:

`https://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct`

This also has links to the relevant University pages. You are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put any such work in a source code repository then you must set access permissions appropriately, limiting access to at most yourself and members of the ILP course team.

— ◇ —

The Informatics Large Practical is not a group practical, so all work that you submit for assessment must be your own, or be acknowledged as coming from a publicly-available source such as Mapbox GeoJSON sample projects, answers posted on StackOverflow, or open-source projects hosted on GitHub, GitLab, BitBucket or elsewhere.

### A.2 Late submission policy

It may be that due to illness or other circumstances beyond your control that you need to submit work late. Submissions which arrive after the coursework deadline will be subject to the School's late submission penalties as detailed at

`https://web.inf.ed.ac.uk/infweb/student-services/taught-students/information-for-students/information-for-all-students/your-studies/late-coursework-extension-requests`.