Manchester Metropolitan University

Department of Computing and Mathematics

ASSIGNMENT COVER SHEET

Unit code and title:	Advanced Programming
Assignment set by:	K. Welsh
Assignment ID: *(e.g. 1CWK50)*	1CWK100
Assignment weighting:	Hackathon Assessment
Assignment title:	100%
Type: (Group/Individual)	Individual
Hand-in deadline:	21:00 9th Jan 2023
Hand-in format and mechanism:	Online, via Moodle

Learning outcomes being assessed:

LO1: Develop moderately complex software solutions meeting a defined specification applying appropriate software architectures, coding and documentation techniques.
LO2: Synthesise documentation and code examples for existing libraries or services to apply standard solutions to common programming problems.
LO3: Use standard software testing techniques to verify and demonstrate the correctness of their code.

Note: it is your responsibility to make sure that your work is complete and available for marking by the deadline. Make sure that you have followed the submission instructions carefully, and your work is submitted in the correct format, using the correct hand-in mechanism (e.g. Moodle upload). If submitting via Moodle, you are advised to check your work after upload, to make sure it has uploaded properly. Do not alter your work after the deadline. You should make at least one full backup copy of your work.

Penalties for late hand-in: see Assessment Regulations for Undergraduate/Postgraduate Programmes of Study on the Student Life web pages. The timeliness of submissions is strictly monitored and enforced.

All coursework has a late submission window of 7 days (i.e. 5 working days), but any work submitted within the late window will be capped at 40%, unless you have an agreed extension. Work submitted after the 7-day late window will be capped at zero, unless you have an agreed extension. See below for further information on extensions.

Please note that individual tutors are unable to grant extensions to coursework.

Extensions: For most coursework assessments, you can request a 7-day extension through Moodle. This will not apply to in-class tests, presentations, interviews, etc, that take place at a specific time. If you need a longer extension you can apply for an Evidenced Extension. For an Evidenced Extension you **MUST** submit 3rd party evidence of the condition or situation which has negatively impacted on your ability to submit or perform in an assessment.

Plagiarism: Plagiarism is the unacknowledged representation of another person's work, or use of their ideas, as one's own. Manchester Metropolitan University takes care to detect plagiarism, employs plagiarism detection software, and imposes severe penalties, as outlined in the Student Handbook (http://www.mmu.ac.uk/academic/casqe/regulations/docs/policies_regulations.pdf and Regulations for Undergraduate Programmes (https://www.mmu.ac.uk/academic/casqe/regulations/assessment/docs/ug-regs.pdf). Bad referencing or submitting the wrong assignment may still be treated as plagiarism. If in doubt, seek advice from your tutor.

If you are unable to upload your work to Moodle: If you have problems submitting your work through Moodle you can email it to the Assessment Team's Contingency Submission Inbox using the email address submit@mmu.ac.uk. You should say in your email which unit the work is for, and ideally provide the name of the Unit Leader. The Assessment team will then forward your work to the appropriate person. If you use this submission method, your work must be emailed by the published deadline, or it will be logged as a late submission. Alternatively, you can save your work to your university OneDrive and submit a Word document to Moodle which includes a link to the folder. It is your responsibility to make sure you share the OneDrive folder with the Unit Leader.

As part of a plagiarism check, you may be asked to attend a meeting with the Unit Leader, or another member of the unit delivery team, where you will be asked to explain your work (e.g. explain the code in a programming assignment). If you are called to one of these meetings, it is very important that you attend.

Assessment Criteria:	Indicated in the attached assignment specification.
Formative Feedback:	In person, via laboratory support on the day of the hackathon.
Summative Feedback Format:	Marks & feedback will be returned via Moodle, within 4 weeks of your submission.

Advanced Programming

Hackathon Task: Transport Web Service

Deadline: 21:00 9th Jan 2023

Introduction

The UK Government provides a DataSet named NaPTAN (National Public Transport Access Nodes) which contains all of the locations at which you can get on or off public transport in the UK [1]. The dataset includes details of bus stops, railway stations, urban/suburban light railway stations (e.g. Manchester's Metrolink and the London Underground), airports and ferry terminals. This is the dataset that mapping apps use to pinpoint the location of public transport.

The dataset is made available in either XML or CSV-based formats by the government [2], but I have created an SQLite database with a trimmed down selection of columns, and a number of near-duplicate records removed for you to use in this hackathon task.

You will create your web service using the Spark Java [3] microservice framework, the SQLite JDBC driver [4], the reference JSON parser [5], and Java's built-in DOM XML parser [6], as you used in the lab sessions for Advanced Programming. **DO NOT** use any other libraries on this assessment: code written using other libraries will not be marked. You can download a starter project, in which the database, libraries, and a small amount of starter code are already set-up, from Moodle.

The Starter Project

Importing

The starter code is available on Moodle as an eclipse project, exported as a zip file. You can import it into your eclipse workspace by selecting File → Import, and then selecting "Existing Projects Into Workspace" from the "General" category. You will see a dialog similar to that depicted in Fig. 1, below.

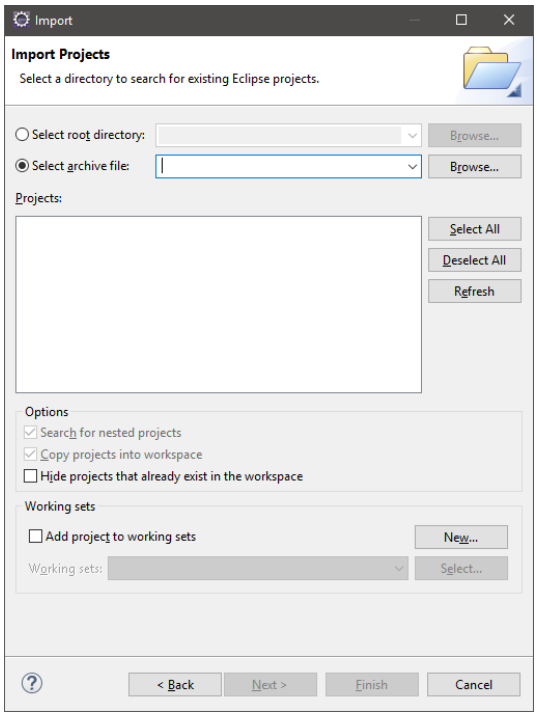

Figure 1: Eclipse Import Projects Dialog

On the import dialog, you should make sure "Select archive file" is selected, then browse to the zip you downloaded from Moodle. Make sure that the "TransportWebService" project is ticked, before clicking the "Finish" button at the bottom of the dialog. You will see the project in the left-hand panel in eclipse, along with your previous projects.

Project Structure

As well as the regular *src* folder, in which your project's source code resides, the starter project has a *lib* folder and a *data* folder. The *lib* folder contains the reference JSON parsing library [5], the SQLite JDBC Driver [4], and the Spark Java microservice framework [3], all of which you have experience using from the Advanced Programming lab exercises and homework sheets. As stated before, you **must not** use any other libraries in the creation of your web service. Code using other libraries will not be marked. The *data* folder contains the SQLite database of the public transport access nodes, which you will query to respond to incoming service requests.

The starter project also contains a small amount of code to get you started, and to allow you to test that everything is working correctly. If you wish, you can add further classes to the project, but for ease of marking I'd ask that you keep the existing *TransportWebService* class and use its *main()* method as the entry point for your web service. The two provided classes are discussed in the following subsections.

TransportWebService Class

This class contains the main entry point for the web service, a standard *main()* method. The imports necessary to use the Spark Java microservice framework [3] are already in place, and there is a simple Route already defined that, when accessed via a web browser, will display the number of entries in the database. You should see approximately 378,000 if all is working properly. To access this

endpoint, you can use the URL http://localhost:8088/test assuming that the server is running and that you haven't changed the port number.

DB Class

This class contains code to manage access to the SQLite database. The existing code automatically connects to the database when an instance of the DB class is created, and disconnects when (or if) the *close()* method is called. The DB class implements the *AutoCloseable* interface, which means that you can use the class inside a try-with-resources block if desired. The DB class also has a simple method to retrieve the number of entries in the database, and this is where the data in the existing TransportWebService Route originates.

The SQLite Database

The database contains just a single table, "Stops", with columns as depicted in Table I, below.

Table I: Public Transport database Stops table structure

Column Name	Column Type	Example of Data
NaptanCode	TEXT	mangpjmt
CommonName	TEXT	Oxford Road Station
Landmark	TEXT	Palace Hotel
Street	TEXT	Oxford Road
Indicator	TEXT	Stop B
Bearing	TEXT	SE
LocalityName	TEXT	Manchester City Centre
Longitude	REAL	-2.240428
Latitude	REAL	53.47403
StopType	TEXT	BUS
BusStopType	TEXT	MKD
TimingStatus	TEXT	OTH

The most important columns in the *Stops* table are the *CommonName, LocalityName, Longitude, Latitude,* and *StopType* columns. The *CommonName* is the name of the station, as it will appear in your web service's output. The *LocalityName* column tells us where the station is, although unfortunately the values for this column are free-form and don't correspond to town/city names directly in all cases. The *Longitude* and *Latitude* columns define the stop's location, which can be used to pinpoint the stop on a map, or to help find the nearest stops to a particular location. The *StopType* column tells us what type of public transport can be caught at this location. I have **not** used the type abbreviations that came with the original data, instead adopting some more readable values. The possible values of the StopType column are depicted in Table II, below.

Table II: StopType Column Values

Type Code	Explanation
BUS	A bus stop
RLW	A railway station
MET	A metro station (e.g. Metrolink, London Underground)
FER	A ferry terminal
AIR	An airport
TXR	A taxi rank

You may wish to browse the SQLite database to get a feel for the structure of the data, and to see some typical rows. This can be performed using any of the standard SQLite tools available, including the one you used in the laboratory sessions earlier in the unit: DB Browser [7].

Number of Stops in Locality (40%)

Your web service should provide a */stopcount* route, which will allow clients to retrieve the number of stops in a particular locality. To achieve this, your server will need to decode a *locality* parameter from the URL query string, similar to the way you completed the "colour" web service exercise in labs (Lab 22, Task 2). Examples of valid URLs that your web service will need to handle are:

```
http://localhost:8088/stopcount?locality=Manchester+City+Centre
http://localhost:8088/stopcount?locality=Wimbledon
http://localhost:8088/stopcount?locality=Hull
```

Your web service will simply return a plaintext number, with no other information for these requests. If the user supplies a locality that doesn't exist, your web service should return a 0. If, however, they omit a value for the *locality* parameter or omit the parameter entirely, the service should return "Invalid Request" as a plaintext string.

Because the user will be supplying the locality that will form part of your SQL query, your web service will need to use a *parametrised query,* as discussed in the JDBC lecture. If you don't, your web service might be vulnerable to an SQL injection attack, making it possible to break the web service or database by manipulating the *locality* parameter. The code to run the query and retrieve the results will be similar to your work on the JDBC lab exercises (Lab 17, Tasks 1-3). Fig. 2, below, depicts the correct SQL query to use.

```
SELECT COUNT(*) AS Number
FROM Stops
WHERE LocalityName = ?;
```

Figure 2: SQL Query to Retrieve Number of Stops in Specified District

Stops By Locality and Type (25%)

Although finding how many stops there are in a locality can be useful, it will be far more common for users to wish to find out the details of all the stops in a locality of a particular transport type. To achieve this, you will create a *stops* route in your web service, with two URL parameters controlling the locality in which to search, and the type of stations to retrieve. Some valid example URLs for the *stops* route are listed below.

```
http://localhost:8088/stops?locality=Manchester+City+Centre&type=MET
http://localhost:8088/stops?locality=Westminster&type=BUS
http://localhost:8088/stops?locality=Tiree&type=FER
```

The SQL query to retrieve the details of the stops for these queries remains fairly simple, although you will need two query parameters this time. The required SQL query is depicted in Fig. 3, below.

```sql
SELECT *
FROM Stops
WHERE LocalityName = ?
AND StopType = ?;
```

Figure 3: SQL Query to Retrieve Stops by Locality & Type

The */stops* route will output data in a JSON-based format, with a JSON array containing a JSON object for each returned result. The JSON Object for each result has *name, locality,* and *type* properties, plus a *location* property, which should itself be a JSON object. The *location* property's JSON object should comprise *Street, Landmark, Indicator* and *Bearing* properties. The values of the properties should be set with the values of the corresponding column in the database, although names do not match exactly. If there are any null values in the retrieved rows, the corresponding properties should be set to an empty string. An example of the output is depicted in Fig. 4, below.


```
[
  {
    "name": "Drumdewan Road End",
    "locality": "Dull",
    "location": {
      "indicator": "at",
      "bearing": "E",
      "street": "B846",
      "landmark": ""
    },
    "type": "BUS"
  },
  {
    "name": "Drumdewan Road End",
    "locality": "Dull",
    "location": {
      "indicator": "opp",
      "bearing": "W",
      "street": "B846",
      "landmark": ""
    },
    "type": "BUS"
  }
]
```


Figure 4: */stops* JSON Output

Your web service **does not** need to format its JSON output over multiple lines, or indent elements. The formatting in Fig. 4, above, is purely for readability. Clients are able to parse JSON data in an unformatted string, and this is how the required JSON parsing library (included with the starter code, same library used in labs) produces output.

The */stops* route should send an appropriate Content-Type HTTP header (application/json) to inform clients that it will be responding with JSON data. If you have set this correctly, your browser may apply some formatting to the JSON returned from the web service when testing, although not all browsers include this feature. You can check that the correct header is being sent in the "Network" tab of your browser's development tools.

In the event of no stops being found in the specified locality of the specified type, then your web service should return an empty JSON array. If the user omits one of the parameters or values from the query, or if the user searches for a station type other than those listed in Table II, the web service should return the string "Invalid Request", which is the same as in the previous section.

Stops By Location (25%)

Unfortunately, the NaPTAN data doesn't do a good job of helping us to find all of the public transport in a particular town or city. The original dataset [1] does contain a *Town* column, but no stops have any value for that particular column. This means that there's no convenient way to search for transport in "Manchester", because the *DistrictName* column for the required entries may read "Ancoats", "Manchester City Centre", or "Hulme" (among others). The government does publish the National Public Transport Gazetteer [8], but this doesn't help either because the localities listed there do not have a reliable town/city value either! Thus, the most useful way to query the dataset is to find the stops closest to a particular location.

Your web service will need a */nearest* route, which will accept three parameters: *latitude, longitude* and *type*. The *latitude* and *longitude* parameters specify the location that the user wishes to search from, while the *type* parameter (as previously) specifies what type of public transport to search for. Phones with GPS provide user locations as standard latitude and longitude values, so a mobile app could readily use the service to find nearby transport. Some example URLs are listed below.

```
http://localhost:8088/nearest?latitude=53.472&longitude=-2.244&type=MET
(My office)

http://localhost:8088/nearest?latitude=51.907&longitude=-0.084&type=RLW
(Village of "Nasty" in Hertfordshire)

http://localhost:8088/nearest?latitude=57.284&longitude=-5.720&type=FER
(The Kyle of Lochalsh in Scotland)
```

The SQL query you will need to find the closest stations is complex, calculating an approximation of the distance between the specified location and each stop. The standard way of calculating this is the Haversine formula, sometimes referred to as a great circle distance. Alas, this requires some trigonometric functions that aren't built into most versions of SQLite. Instead, you will use an approximation based on Pythagoras to find the five closest stops, as depicted in Fig. 5, below.

```sql
SELECT *
FROM Stops
WHERE
  StopType = "MET"
  AND Latitude IS NOT NULL
  AND Longitude IS NOT NULL
ORDER BY
  (
    ((53.472 - Latitude) * (53.472 - Latitude)) +
    (0.595 * ((-2.244 - Longitude) * (-2.244 - Longitude)))
  )
  ASC
LIMIT 5;
```

Figure 5: SQL Query to Retrieve the Five Stops Nearest to My Office

The SQL query depicted in Fig. 5, above uses the approximate location of my office for illustration. You will need to use a parametrised query, as you did in the previous sections. The query's WHERE clause specifies the station type and excludes rows in the database that do not have a latitude and longitude value. The ORDER BY clause calculates the difference between the supplied latitude (53.472) and the latitude of each stop in the database, and squares it. The same is done for the longitude (-2.244) – but a multiplier is used (0.595) because one degree of longitude is equal to one degree of latitude only at the equator. The multiplier's value is the cosine of the latitude, calculated separately. This assumes that the earth is spherical, which isn't totally accurate, but the calculation provides a reasonable approximation. We omit the square root normally needed for Pythagoras: we don't need to calculate an exact distance, just order the rows.

Your web service will return the details of the nearest stops in an XML-based format, with a <NearestStops> element containing one <Stop> element for each of the results returned from the database. Each <Stop> element will have a *code* attribute, set to the value of the database's *NaptanCode* column for that row. Each <Stop> element will also contain <Name> and <Locality> elements with the corresponding values from the database, as well as a <Location> sub-element. Each stop's <Location> sub-element will contain <Street>, <Landmark>, <Latitude>, and <Longitude> elements, each with appropriate values from the database. An example showing the correct output format for three Metrolink stations is depicted in Fig. 6, below. Your output will always contain five stations, Fig. 6 depicts only three results merely for brevity.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<NearestStops>
    <Stop code="mantmtwg">
        <Name>Deansgate-Castlefield (Manchester Metrolink)</Name>
        <Locality>Manchester City Centre</Locality>
        <Location>
            <Street>Off Whitworth Street West</Street>
            <Landmark>Manchester Central Convention Complex</Landmark>
            <Latitude>53.47478743755</Latitude>
            <Longitude>-2.25036146685</Longitude>
        </Location>
    </Stop>
    <Stop code="mantmwdt">
        <Name>St Peter's Square (Manchester Metrolink)</Name>
        <Locality>Manchester City Centre</Locality>
        <Location>
            <Street>Mosley Street</Street>
            <Landmark>Manchester Central Library</Landmark>
            <Latitude>53.47824514078</Latitude>
            <Longitude>-2.24304399517</Longitude>
        </Location>
    </Stop>
    <Stop code="mantmwam">
        <Name>Piccadilly Gardens (Manchester Metrolink)</Name>
        <Locality>Manchester City Centre</Locality>
        <Location>
            <Street>Parker Street</Street>
            <Landmark>The Portland</Landmark>
            <Latitude>53.48027969905</Latitude>
            <Longitude>-2.23699823251</Longitude>
        </Location>
    </Stop>
</NearestStops>
```

Figure 6: */nearest* Route XML Output Format

Because your web service is returning XML data, you should send the client an HTTP header telling them to expect XML data. The correct value of the Content-Type header for XML data is application/xml. If you have set it correctly, your browser may format the returned data when testing, although not all browsers support this feature. You can check that the correct header is being sent in the "Network" tab of your browser's development tools. As in the previous section, you do not need to split your output over multiple lines or indent the returned XML.

Code Quality (10%)

You are being assessed not only on how much functionality you are able to implement during the hackathon, but also on how robust, performant, maintainable and secure the code you create is. In industry, you will frequently be working under time pressure, and code that works but is of poor quality will cause problems for yourself and others.

The web service's code should be structured using appropriate classes and methods so as to avoid any significant code duplication, and to maximise ease of reading and maintenance. You should try to separate code that concerns the web service itself (e.g. parameter and HTTP handling), the database

(e.g. queries and results), and the encoding of data in responses (e.g. JSON and XML building) to achieve a degree of encapsulation and promote separation of concerns. The code should be commented appropriately, using the JavaDoc standard. You should also use parametrised queries where appropriate when accessing the database to avoid SQL injection vulnerabilities. As stated previously, you must not use any other libraries than those included with the starter code.

Support

Support will be available from 9:00 until 12:00 noon, and from 13:00 to 18:00, in person, in the C2.05 lab in the John Dalton building. You are welcome to work there, in one of the surrounding labs, or at home if you prefer. If working from off campus, do be aware that tutors will not be answering e-mails or teams messages during the hackathon. Tutors will be in the labs, offering support to those present, and unable to access other forms of communication.

Submission and Marking

Exporting Your Work

You will need to export your from eclipse for submission, zipping the project up into a single file. The easiest way to achieve this is to use the File → Export menu in eclipse, selecting *General* and then *Archive File*. Once you have selected this, a dialog similar to that depicted in Fig. 7, below, appears.

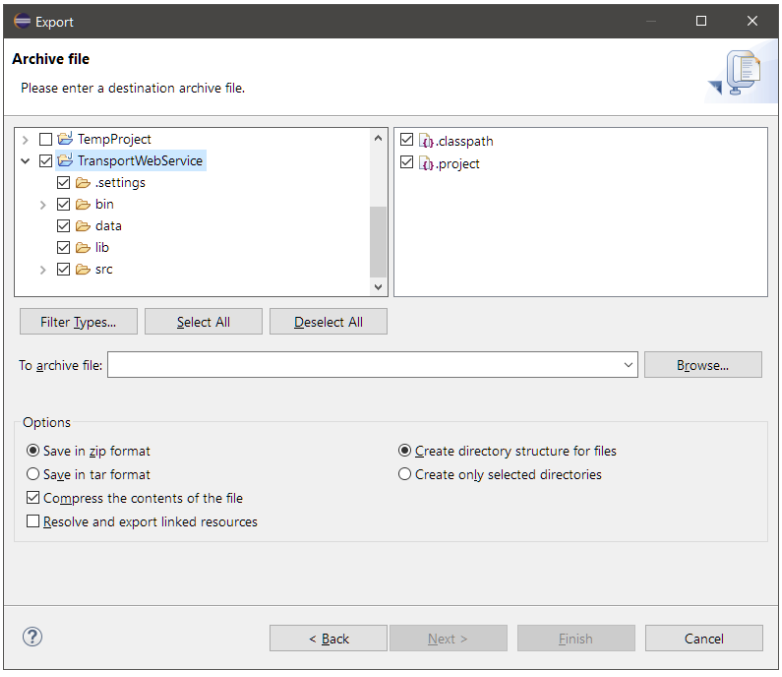

Figure 7: Eclipse Zip Export Dialog

On the dialog depicted in Fig. 7, above, you should make sure that the checkbox to the left of the TransportWebService project is ticked. Ticking this checkbox ensures that your project is exported in its entirety: code, data, libraries and all. You will also need to ensure that the *Compress the contents of the file* checkbox is ticked, so that the file will be small enough to upload to Moodle.

Submitting Your Work

Submissions will be made via Moodle, using the link in the *Assessment* block of the Advanced Programming Moodle page. You must upload your work before the deadline, which is 21:00 today. Leave plenty of time to upload your work to Moodle, your exported project may be quite large and need time to upload, particularly if Moodle gets busier as the deadline approaches.

Work that is submitted late, but within the next week, will be subject to a 40% mark cap. Work submitted more than a week late will be awarded a mark of zero. This assessment has been exempted from the self-certification process for exceptional factors; you cannot grant yourself a week's extension. Students who are unable to participate in the hackathon assessment and are granted exceptional factors extension via the evidence-based process, or who cannot participate in the hackathon on the grounds of a disability, will be set an alternative assessment.

Marking Criteria

You are being graded on both *how much* you can achieve on the day, and *how good* your implemented work is. Your web service will be tested to find out how closely it meets this specification, and your code will be reviewed to assess how robust, maintainable and secure your implementation is. The marking scheme depicted in Fig. 8, overleaf, provides more detailed guidance.

Criterion	Bad Fail	Marginal Fail	Pass	II(II)	II(I)	I	Distinctive
Number of Stops in Locality (40%)	Little or no progress towards returning number of stops in given locality	Some progress, but correct number is not returned for valid queries	Correct number returned, but stop count endpoint doesn't properly meet specification	Stop count endpoint largely meets spec, with some minor issue(s), possibly only in some specific cases (e.g. invalid queries not handled correctly)		Stop count endpoint fully meets specification	
Stops by Locality & Type (25%)	Little or no progress towards returning stops by locality and type	Some progress, but correct data is not returned for valid queries	Correct stop data returned, but stops endpoint doesn't properly meet specification	Stops endpoint largely meets spec, with some minor issue(s), possibly only in some specific cases	Stops endpoint meets spec, but fails to send correct Content-Type header, but is otherwise correct	Stops endpoint fully meets specification	
Stops by Location (25%)	Little or no progress towards returning stops nearest specified location	Some progress, but stop data is not returned for valid queries	Stop data is returned for valid queries, but the nearest stops are not correctly identified	Correct stop data is returned, with some minor issue(s), possibly only in some specific cases (e.g. parameter validation, incorrect output schema)		Nearest endpoint meets spec, but fails to send correct Content-Type header	Nearest endpoint fully meets specification
Code Quality (10%)	Serious and/or systemic problems with code quality	Minor problem or problems with code quality	Code is engineered, structured and commented acceptably	Code is engineered well, with little repetition. Code is readable and maintainable, with appropriate JavaDoc comments for most classes & methods		Code is very well engineered, providing an elegant and clean solution. JavaDoc & Inline comments are comprehensive, considered and helpful	

Figure 8: Marking Scheme

Plagiarism, Collusion & Duplication of Material

I, the Faculty, and the University all take academic malpractice very seriously. The work you submit for this assignment must be your own, completed without any significant assistance from others. Be particularly careful when helping friends to avoid them producing work similar to your own. I will be running all submitted work through an automated plagiarism checker, and I am generally vigilant when marking. The penalties for academic malpractice can be severe. Please refer to the guidance at https://www.mmu.ac.uk/student-case-management/guidance-for-students/academic-misconduct/ for further information.

References

[1] Department for Transport, 'National Public Transport Access Nodes (NaPTAN)', Aug. 23, 2022. https://www.data.gov.uk/dataset/ff93ffc1-6656-47d8-9155-85ea0b8f2251/national-public-transport-access-nodes-naptan (accessed Sep. 07, 2022).

[2] Department for Transport, 'NaPTAN user guide', *GOV.UK*. https://www.gov.uk/government/publications/national-public-transport-access-node-schema/html-version-of-schema (accessed Sep. 07, 2022).

[3] P. Wendel, 'Spark Framework: An expressive web framework for Kotlin and Java'. https://sparkjava.com/ (accessed Sep. 07, 2022).

[4] T. L. Saito, 'SQLite JDBC Driver'. Sep. 04, 2022. Accessed: Sep. 07, 2022. [Online]. Available: https://github.com/xerial/sqlite-jdbc

[5] S. Leary, 'JSON in Java [package org.json]'. Sep. 06, 2022. Accessed: Sep. 07, 2022. [Online]. Available: https://github.com/stleary/JSON-java

[6] Oracle Corporation, 'Introduction to JAXP - Java API for XML Processing (JAXP) Tutorial'. https://www.oracle.com/java/technologies/jaxp-introduction.html (accessed Sep. 07, 2022).

[7] M. Kleusberg, 'DB Browser for SQLite'. https://sqlitebrowser.org/ (accessed Sep. 07, 2022).

[8] Department for Transport, 'National Public Transport Gazetteer (NPTG)', Aug. 23, 2022. https://www.data.gov.uk/dataset/3b1766bf-04a3-44f5-bea9-5c74cf002e1d/national-public-transport-gazetteer-nptg (accessed Sep. 07, 2022).