Manchester Metropolitan University

Department of Computing and Mathematics

ASSESSMENT COVER SHEET 2023/24

Unit Code and Title:	6G5Z0041 – Advanced Programming
Assessment Set By:	K. Welsh
Assessment ID:	1CWK100
Assessment Weighting:	100%
Assessment Title:	Hackathon
Type:	Individual
Hand-In Deadline:	See Moodle
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. If submitting via OneDrive, ensure that your tutors have access to the work. Do not alter your work after the deadline. You should make at least one full backup copy of your work.

Penalties for late submission

The timeliness of submissions is strictly monitored and enforced.

All coursework has a late submission window of 7 calendar 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 'Assessment Mitigation' below for further information on extensions.

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

Assessment Mitigation

If there is a valid reason why you are unable to submit your assessment by the deadline you may apply for assessment mitigation. Only evidenced extension requests are allowed on this assessment. Moodle (in the 'Assessments' block on the right-hand side of the page):

- **Self-certification**: does **not** require you to submit evidence. It allows you to add a short extension to a deadline. This is not available for event-based assessments such as in-class tests, presentations, interviews, etc. Self-certification extensions are **not** available on this assessment.

- **Evidenced extensions:** requires you to provide independent evidence of a situation which has impacted you. Allows you to apply for a longer extension and is available for event-based assessment such as in-class test, presentations, interviews, etc. For event-based assessments, the normal outcome is that the assessment will be deferred to the Summer resit period.

Further information about Assessment Mitigation is available on the dedicated Assessments page: https://www.mmu.ac.uk/student-life/course/assessments#ai-69991-0

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 Code of Conduct and Regulations for Undergraduate Programmes. Poor referencing or submitting the wrong assignment may still be treated as plagiarism. If in doubt, seek advice from your tutor.

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.

If you are unable to upload your work to Moodle

If you have problems submitting your work through Moodle, there is a Contingency Submission Form on the university's Assist ticketing system, where you can upload your work. If you use this submission method, your work must be uploaded **by the published deadline**, or it will be logged as a late submission. Alternatively, you can save your work into a single zip folder then upload the zip folder 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, or it will not be possible to mark your work.**

Assessment Regulations

For further information see Assessment Regulations for Undergraduate/Postgraduate Programmes of Study on the Student Life web pages.

Formative Feedback:	In person, via laboratory support on the day of the hackathon.
Summative Feedback:	Marks & feedback will be returned via Moodle, within 4 weeks of your submission.

Advanced Programming

Hackathon Task: Earthquake Web Service

Deadline: 21:00 16th January 2024

Introduction

There are many earthquakes each year and around the world, even though not all of them cause significant danger or widespread property damage. There are a number of scientific institutions worldwide who monitor seismic activity to pinpoint the source and magnitude of earthquakes, including both the British [1] and US Geological Survey [2] organisations (BGS and USGS, respectively). The USGS publish data on all known earthquakes and other significant seismic activity [3], enabling researchers to monitor trends and verify findings.

Using the USGS data, I have created a SQLite database of all the earthquakes worldwide since the year 1900 with a magnitude of at least 5.0 on the Richter scale. I omitted earthquakes that were smaller smaller or older to limit the size of the database. Your task is to create a web service that allows users to retrieve data about the earthquakes in the database.

You will create your web service using the Spark Java [4] microservice framework, the SQLite JDBC driver [5], the reference JSON parser [6], and Java's built-in DOM XML parser [7], 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 should download the 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.

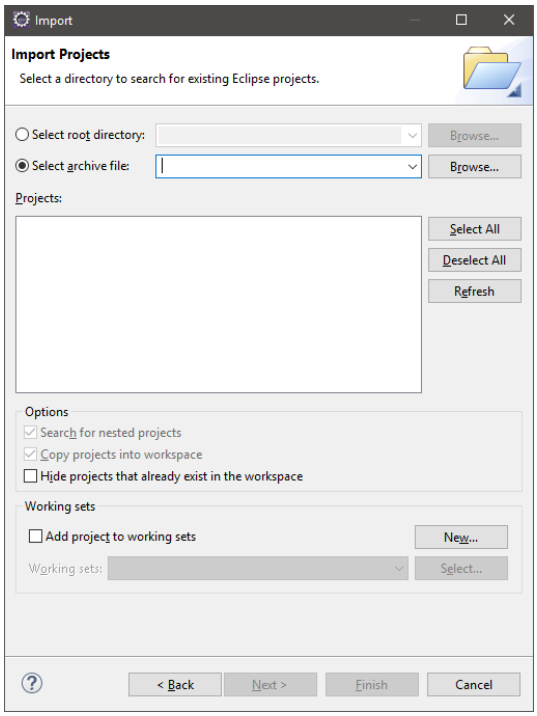

Figure 1: Eclipse Import Projects Dialog

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 "EarthquakeWebService" 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 [6], the SQLite JDBC Driver [5], and the Spark Java microservice framework [4], all of which you have experience using from the Advanced Programming lab exercises. 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 *EarthquakeWebService* class and use its *main()* method as the entry point for your web service. The two provided classes are discussed in the following subsections.

EarthquakeWebService 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 [4] 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 98,000 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 *EarthquakeWebService* Route originates.

The SQLite Database

The database contains just a single table, "earthquakes", with a fair number of columns. You will not be using all of the columns in the table, but the key ones are depicted in Table I, below.

Table I: Database's earthquake table structure

Column Name	Column Type	Example of Data
time	TEXT	2022-12-31T03:31:43.824
latitude	REAL	-23.1061
longitude	REAL	-68.8988
depth	REAL	96.289
mag	REAL	5.1
id	TEXT	us7000j15y
place	TEXT	71 km S of Calama, Chile
type	TEXT	earthquake

The *time* column contains an ISO8601-formatted date and time which you will be able to parse using the LocalDateTime.parse() method. The *longitude* and *latitude* columns define the location of the earthquake's epicentre on the earth's surface, while the *depth* column specifies how far below the earth's surface it occurred – together giving the *hypocentre*. You can use the *latitude* and *longitude* columns to find the location on a map, or to find the closest earthquakes to a particular location. The *mag* column specifies the earthquake's magnitude – note the database does not contain any earthquakes smaller than 5.0. The *place* column contains a short human-readable description of the epicentre's location, while the *type* column specifies whether the measured seismic activity was caused by an earthquake or something else. Large explosions can sometimes be detected, as can nuclear weapons tests.

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 [8]. There is also documentation on what the columns mean available from the USGS [9].

Number of Earthquakes with Minimum Magnitude (40%)

Your web service should provide a */quakecount* route, which will allow clients to retrieve the number of earthquakes in the database with at least the specified magnitude. Smaller earthquakes are more common than larger ones, and this endpoint allows users to get a feel for the comparative rarity of earthquakes of a given magnitude. To achieve this, your server will need to decode a *magnitude* 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/quakecount?magnitude=5.5
http://localhost:8088/quakecount?magnitude=6.0
http://localhost:8088/quakecount?magnitude=7.1
```

Your web service will simply return a plaintext number, with no other information for these requests. If the user supplies a magnitude larger than any entries in the database, your web service should return 0. If, however, they supply an invalid value, omit a value for the *magnitude* parameter or omit the parameter entirely, the service should return "Invalid Magnitude" as a plaintext string.

Because the user will be supplying the magnitude 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 *magnitude* 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 earthquakes
WHERE mag >= ?;
```

Figure 2: SQL Query to Retrieve Number of Earthquakes of the Specified Magnitude or Larger

Earthquakes by Year and Magnitude (25%)

Although finding how many earthquakes there have been of a particular magnitude can be useful, it will be far more common for users to wish to find out the details of specific earthquakes. To achieve this, you will create a *quakesbyyear* route in your web service, with two URL parameters controlling the year in which to search, and the minimum magnitude as in the previous section. Some valid example URLs for the *quakesbyyear* route are listed below.

```
http://localhost:8088/quakesbyyear?year=2022&magnitude=6.5
http://localhost:8088/quakesbyyear?year=1994&magnitude=6.0
```

The SQL query to retrieve the details of the earthquakes 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. Note that you will need to add a % wildcard to the value you bind to the *time* parameter.

```sql
SELECT *
FROM earthquakes
WHERE time LIKE ?
AND mag >= ?
ORDER BY time ASC;
```

Figure 3: SQL Query to Retrieve Eartquakes by Year & Magnitude

The */quakesbyyear* 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 will have *id, date, time,* and *magnitude* properties, plus a *location* property which should itself be a JSON object. The *location* property's JSON object should comprise *latitude, longitude,* and *description* properties. The values of the properties should be set with the values from the appropriate columns in the database, although the names in the JSON object to not exactly match those in the database table. If any of the result rows have a null value for a particular property, set it to an empty string for TEXT columns or a zero for numeric columns. An example of the output for two earthquakes is depicted in Fig. 4, below.


```json
[
  {
    "date": "1999-01-19",
    "magnitude": 7,
    "location": {
      "latitude": -4.596,
      "description": "110 km ESE of Kokopo, Papua New Guinea",
      "longitude": 153.235
    },
    "id": "usp00091ag",
    "time": "03:35:33.840"
  },
  {
    "date": "1999-02-06",
    "magnitude": 7.3,
    "location": {
      "latitude": -12.853,
      "description": "146 km NW of Sola, Vanuatu",
      "longitude": 166.697
    },
    "id": "usp00092cr",
    "time": "21:47:59.470"
  }
]
```


Figure 4: */quakesbyyear* 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, the same library used in labs) produces output.

The */quakecount* 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 earthquakes being found in the specified year of the specified magnitude, 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 supplies an invalid value for either parameter, the web service should return a plaintext error string. If the *year* parameter is invalid the web service should return "Invalid Year". If the *magnitude* parameter is invalid, the web service should return "Invalid Magnitude". If both parameters are invalid, you only need to return the message for the invalid value first encountered.

Earthquakes By Location (25%)

Users are likely to be interested in the earthquakes that have happened closest to them. Fortunately, the database contains the latitude and longitude of each earthquake, so location-based searches are possible. Your web service will need a */quakesbylocation* route, which will accept three parameters: *latitude, longitude* and *magnitude*. The *latitude* and *longitude* parameters specify the location that the user wishes to search from, while the *magnitude* parameter (as previously) specifies the minimum magnitude of earthquake to search for. Some example URLs are listed below.

```
http://localhost:8088/quakesbylocation?latitude=53.472&longitude=-2.244&magnitude=6.0
(My office)

http://localhost:8088/quakesbylocation?latitude=29.975&longitude=31.1375&magnitude=7.5
(Great Sphinx of Giza)

http://localhost:8088/quakesbylocation?latitude=32.277&longitude=-64.797&magnitude=6.5
(Mt Pleasant, Bermuda)
```

The SQL query you will need to find the closest earthquakes is complex, calculating an approximation of the distance between the specified location and each earthquake's epicentre. 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 ten closest earthquakes, as depicted in Fig. 5, below.

```sql
SELECT *
FROM earthquakes
WHERE
  mag >= 6.5
ORDER BY
  (
    ((53.472 - Latitude) * (53.472 - Latitude)) +
    (0.595 * ((-2.244 - Longitude) * (-2.244 - Longitude)))
  )
  ASC
LIMIT 10;
```

Figure 5: SQL Query to Retrieve the Earthquakes 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 minimum magnitude, as in the earlier sections of the assessment. The ORDER BY clause calculates the difference between the supplied latitude (53.472) and the latitude of each earthquake 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 earthquakes in an XML-based format, with a <Earthquakes> element containing one <Earthquake> element for each of the results returned from the database. Each <Earthquake> element will have an *id* attribute, set to the value of the database's *id* column for that row. Each <Earthquake> element will also contain <Date>, <Time> and <Magnitude> elements with the corresponding values from the database, as well as a <Location> sub-element. Each earthquake's <Location> sub-element will contain <Latitude>, <Longitude>, and <Description> elements, each with appropriate values from the database. An example showing the correct output format for three results is depicted in Fig. 6, below. Your output will contain up to ten earthquakes, Fig. 6 depicts only three results for brevity.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Earthquakes>
    <Earthquake id="iscgem898227">
        <Date>1946-01-25</Date>
        <Time>17:31:49.630</Time>
        <Magnitude>6.15</Magnitude>
        <Location>
            <Latitude>46.499</Latitude>
            <Longitude>7.644</Longitude>
            <Description>2 km WNW of Kandersteg, Switzerland</Description>
        </Location>
    </Earthquake>
    <Earthquake id="iscgem872748">
        <Date>1963-07-19</Date>
        <Time>05:45:27.220</Time>
        <Magnitude>6.3</Magnitude>
        <Location>
            <Latitude>43.343</Latitude>
            <Longitude>8.153</Longitude>
            <Description>58 km SSE of San Lorenzo al Mare, Italy</Description>
        </Location>
    </Earthquake>
    <Earthquake id="iscgem872749">
        <Date>1963-07-19</Date>
        <Time>05:46:05.200</Time>
        <Magnitude>6.1</Magnitude>
        <Location>
            <Latitude>43.344</Latitude>
            <Longitude>8.278</Longitude>
            <Description>62 km SSE of San Lorenzo al Mare, Italy</Description>
        </Location>
    </Earthquake>
</Earthquakes>
```

Figure 6: */quakesbylocation* 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.

If the user omits one of the parameters or values from the query, or if any of the parameter values are invalid, the web service should return a plaintext string of "Invalid Latitude", "Invalid Longitude", or "Invalid Magnitude", depending on which value was invalid. If multiple parameters are invalid, you only need to return the message for the invalid value first encountered, as in the previous section.

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 **C-block third floor labs** in the 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.

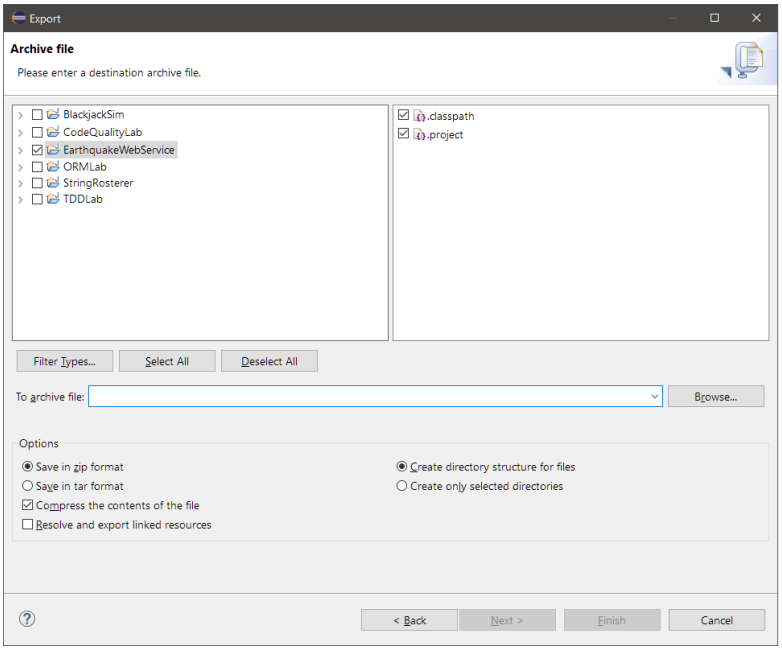

Figure 7: Eclipse Zip Export Dialog

On the dialog depicted in Fig. 7, above, you should make sure that the checkbox to the left of the EarthquakeWebService 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.

You are again reminded that you **must not** use any additional libraries (e.g. jar files) other than those supplied with the starter project. Work implemented using additional libraries or dependencies will not be marked.

Criterion	Bad Fail	Marginal Fail	Pass	II(II)	II(I)	I	Distinctive
Number of Stops in Locality (40%)	Little or no progress towards returning earthquakes of given magnitude	Some progress, but correct number is not returned for valid queries	Correct number returned, but quake count endpoint doesn't properly meet specification	Earthquake count endpoint largely meets spec, with some minor issue(s), possibly only in some specific cases (e.g. invalid queries not handled correctly)		Earthquake count endpoint fully meets specification	
Stops by Locality & Type (25%)	Little or no progress towards returning earthquakes by year and magnitude	Some progress, but correct data is not returned for valid queries	Correct earthquake data returned, but endpoint doesn't properly meet specification	Endpoint largely meets spec, with some minor issue(s), possibly only in some specific cases	Quakes-by-Year endpoint fails to send correct Content-Type header, but is otherwise correct	Quakes-by-Year endpoint fully meets specification	
Stops by Location (25%)	Little or no progress towards returning earthquakes nearest specified location	Some progress, but earthquake data is not returned for valid queries	Earthquake data is returned for valid queries, but correct earthquakes are not correctly identified	Correct earthquake data is returned, with some minor issue(s), possibly only in some specific cases (e.g. parameter validation, incorrect output schema)		Quakes-by-Location endpoint meets spec, but fails to send correct Content-Type header	Quakes-by-Location 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] 'About BGS', *British Geological Survey*. https://www.bgs.ac.uk/about-bgs/ (accessed Jul. 17, 2023).

[2] 'Who We Are | U.S. Geological Survey'. https://www.usgs.gov/about/about-us/who-we-are (accessed Jul. 17, 2023).

[3] 'Earthquakes | U.S. Geological Survey'. https://www.usgs.gov/programs/earthquake-hazards/earthquakes (accessed Jul. 24, 2023).

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

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

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

[7] 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).

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

[9] 'ANSS Comprehensive Earthquake Catalog (ComCat) Event Terms Documentation'. https://earthquake.usgs.gov/data/comcat/data-eventterms.php (accessed Jul. 17, 2023).