Assignment 5: Course Planner

- Submit deliverables to CourSys: https://courses.cs.sfu.ca/
- Late penalty: 10% per calendar day (each 0 to 24 hour period past due), max 2 days late.
- This assignment may be done **individually or in pairs.** Do not show other students your code, do not copy code found online, do not reuse solutions from previous semesters or courses or assignments, and do not post questions about the assignment online. Please direct all questions to the instructor or TA via the help list: cmpt-213-help@sfu.ca;
  - You may use *ideas* (short code snippets, tutorials, documentation...) you find online and from others, but your solution must be your own.
- See the marking guide for details on how each part will be marked.

1. Introduction

Create a Spring Boot server which helps students plan their courses. Specifically, it will show students when a course has been offered in the past which may help predict when it will be offered in the future. You will be writing the server which will use a REST interface to support a web-based UI (web UI provided to you by the instructor).

The server will read a locally stored CSV input file containing data about course offerings at SFU. It will then process the data to organize it into a well structured model. The server's controllers will provide access to this data via a REST API.

The project is divided into two phases, each with its own due date (see website):

1.1 Phase 1:

1. Model reads in data from a provided CSV file.
2. A simple Spring Boot controller can trigger the model to print out the model in a well-structured way to the local terminal (i.e., the server's terminal, not over HTTP back to the calling user!).

1.2 Phase 2:

Add the following features:

1. Support web interface for a web-client (provided by instructor)
2. For a given department, generate data of how many students took classes in that department each semester.
3. REST API interface to add more class offerings, and an ability to record changes made to a specific course (i.e., track what sections have been added to the course).

2. Phase 1 Requirements

Your phase 1 submission *must* meet the following requirements.

2.1 Data Input Requirements

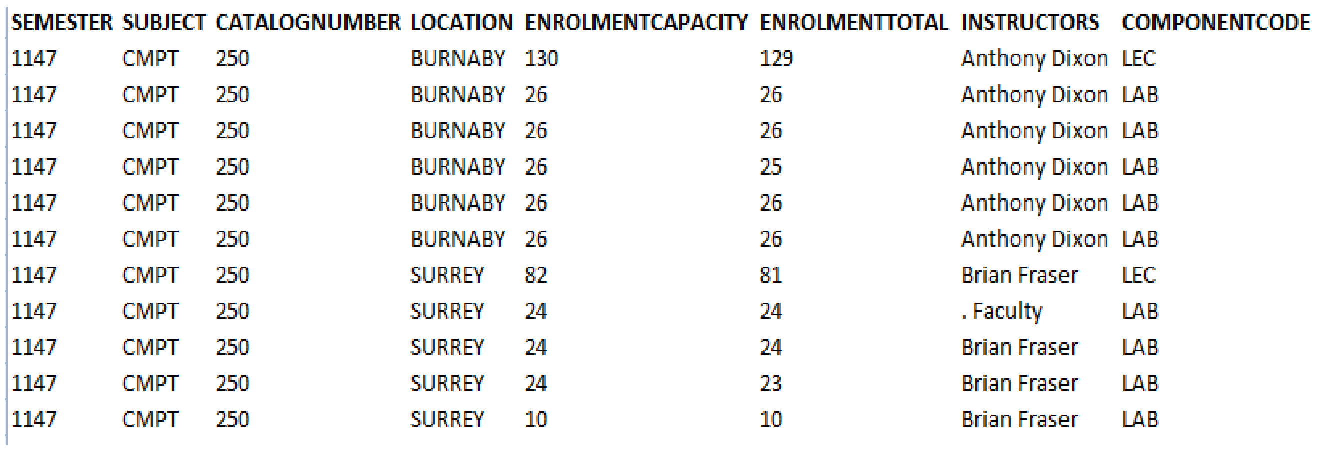
1. Create a folder in your project name `data/` (it will be at the same level as the `src/` folder (not inside it).
2. Copy the provided `course_data_2018.csv` file into this folder. A new data file may become available later.
3. The file is a comma-separated text file with the following columns:
    1. **SEMESTER:**
       The semester number of the offering.
       - The first three digits refer to the year.
       - The last digit is the semester: 1=Spring, 4=Summer, 7=Fall. You need handle only these three semester (i.e., you will not be tested with a semester number 5).
    2. **SUBJECT:**
       The department. Will be a short string.
    3. **CATALOGNUMBER:**
       The course "number". May be a number like 213, or a string like 105W or 1XX.
    4. **LOCATION:**
       The campus:
       - Campuses of interest: `BURNABY`, `SURREY`, `HRBRCNTR`.
       - Other locations: `OFF`, `KIT`, `KAM`, `WML`, …. (anything else).
    5. **ENROLEMENTCAPACITY:**
       The enrollment cap for the section (i.e., maximum number of seats)
    6. **ENROLMENTTOTAL:**
       The enrollment total for the section (i.e., the number of people who took that section).
    7. **INSTRUCTORS:**
       The instructor(s) which taught that section of the course. Comma separated if more than one. "(null)" if unknown (information not stored in the database).
    8. **COMPONENTCODE:**
       The type of section as a short string such as `LEC`, `TUT`, `SEC`, `OPL`, ...
4. At startup, your program must be hard-coded (using a named constant) to load the "`data/course_data_2018.csv`" file.
    1. Your application will only be tested with well formed input files, such as the one provided. We may test with alternative contents, such as different subjects, courses, locations, years, and section types.
    2. You may *not* use a 3rd party CSV file reader.
    3. If the file is not found, or there was an error reading the file or its contents, have the program exit. It is fine to exit due to an uncaught exception, or with a specific error message.

2.2 Required Controller Features

Have a `GET /dump-model` end-point which triggers your application to dump out a summary of the model to the server's terminal. Note that this output is to the server's terminal, not in the client's terminal: the data is not being returned via HTTP, just printed.

Model dump format

- Course number
- Per {semester, location} of that course, display:
  - The semester (SFU semester number), location, and instructor(s).
  - Section type (such as LEC, LAB, …)
  - Enrollment for each section-type: number students enrolled and total capacity.
    1. Sum together instances of a specific section-type within an offering. For example, add up the enrollment and capacity of all labs for a specific course offering.
    2. For example, consider the following lines of the input file:



SEMESTER	SUBJECT	CATALOGNUMBER	LOCATION	ENROLMENTCAPACITY	ENROLMENTTOTAL	INSTRUCTORS	COMPONENTCODE
1147	CMPT	250	BURNABY	130	129	Anthony Dixon	LEC
1147	CMPT	250	BURNABY	26	26	Anthony Dixon	LAB
1147	CMPT	250	BURNABY	26	26	Anthony Dixon	LAB
1147	CMPT	250	BURNABY	26	25	Anthony Dixon	LAB
1147	CMPT	250	BURNABY	26	26	Anthony Dixon	LAB
1147	CMPT	250	BURNABY	26	26	Anthony Dixon	LAB
1147	CMPT	250	SURREY	82	81	Brian Fraser	LEC
1147	CMPT	250	SURREY	24	24	. Faculty	LAB
1147	CMPT	250	SURREY	24	24	Brian Fraser	LAB
1147	CMPT	250	SURREY	24	23	Brian Fraser	LAB
1147	CMPT	250	SURREY	10	10	Brian Fraser	LAB

Figure 1: Sample lines in the input data file.

  - When processed, this generates the following two offerings (Burnaby and Surrey).

```
CMPT 250
      1147 in BURNABY by Anthony Dixon
            Type=LAB, Enrollment=129/130
            Type=LEC, Enrollment=129/130
      1147 in SURREY by Brian Fraser, . Faculty
            Type=LAB, Enrollment=81/82
            Type=LEC, Enrollment=81/82
```

Text 1: Course offering and section information for the lines of data shown in Figure 1.

- The data in the file may not seem to make sense. There could be more people in lecture than lab, there could be sections with 0 capacity, or 0 enrolled, or 9999. This is OK: just process the numbers as though they are valid numbers.
- **Assume there is at most one offering of each course at each campus during a single semester. Therefore aggregate all lectures, all tutorials, …. For example, group together all CMPT 120 information for Burnaby during the summer of 2000, even if there were actually two CMPT 120 sections in Burnaby that semester.**

3. Phase 2 Requirements

You must add a REST API to your program to allow a web front-end (web page) to access your program.

3.1 Curl Commands

- A file of curl commands is provided on the course website which shows the features your back-end system must implement.
  - Note that you'll need to edit the file and change the IDs for the department, course, and offering as these are specific to a given implementation and your IDs will very likely differ from mine.
- You may change the REST API (see description in UI section), but as a starting point, plan to implement the interface as found in this file.
- Sample output is also provided for when the script is run on the latest data.

3.2 UI

- A fully-functional web-UI is provided on the course website. It is designed to connect to a REST API, as explained in the API document, sample curl commands, and sample output.
  - See course website for a video showing the features of the UI when fulling working.
- The provided UI should be extracted into a new public/ folder inside your project (at the root of your project). Once extracted you should be able to run your program and browse to `http://localhost:8080` to access the UI.
- It is expected that you should not need to modify the UI: Once you implement the required REST API end-points functionality of the UI will function.
- You may, if you desire, change the REST API design for your application.
  - If you do this you must also edit the web UI to work with your modified interface.
  - Additionally, you must include a docs/ folder in your project with the curl commands file provided by the instructor that you have modified to interact with your interface.
  - Of specific interest are the commands under the "# error" comments which test how your interface responds to incorrect API requests.
  - The bulk of the processing must be done in the Java back-end (you may not re-write the system to do most of the work in a JavaScript front-end, for example).

4. Coding Requirements

Your program must:

1. Follow the course's code style guide.
2. Separate the controllers from the model.
3. Modularity and class design
    - You must have well design classes.
    - Use multiple classes in your model to store the system's information. You must have at least 5 or more model classes (or even 10+).
4. Your program must not search through the entire data-set each time the user makes a selection on the UI (i.e., via the end-point). Processing all the data each request makes the model too slow to keep up because with a huge data set, it is a *lot* of extra work.
    - Instead, process the data set into meaningful collections of objects, and then access these objects without having to search all records on each click. For example, when loading the data set, process it into courses, and then selecting a course can quickly load the required information.

5. Hints

The following hints may be useful as you create your application. These are not requirements, so you need not follow them. Please see the marking guide for exactly what is being marked.

5.1 Model Design

1. Start by designing the data storage classes.
   - Identify classes by analyzing input data-file format, and needs of the UI.
   - Do CRC cards for your classes.
2. Develop the API (i.e., method calls) the model needs to support in order for the web interface to work.
   - Consider using the facade pattern to expose a single model interface from the model.
3. Think about low level classes you could use. For example, would having a `Semester` class allow you to encapsulate some logic about converting a string ("1037") into a `Semester`? What about a class which can encapsulate reading a CSV file without any knowledge of the meaning of the data?

5.2 General tips

1. Be *very* careful using ==. Most of the time you'll need `.equals()`!
2. Use asserts, and check that they are turned on! (Temporarily add an "assert false;" to test).
3. Considering overriding `equals()` for most of your data classes. This can be very useful when you are searching a list of items to see if it's already in the list.

5.3 Model debug output

- The `/dump-model` end-point displays the entire data set to the console. Use this to check that your read-in and data-processing routines are working. Required output format shown to right.
- Note: You should carefully review the output to ensure your model is functioning.

```
CMPT 106
        1107 in SURREY by Harinder Khangura
                Type=LEC, Enrollment=29/32
                Type=TUT, Enrollment=29/32
        1117 in SURREY by Harinder Khangura
                Type=LEC, Enrollment=30/35
                Type=TUT, Enrollment=30/35
        1131 in SURREY by Harinder Khangura
                Type=LAB, Enrollment=57/61
                Type=LEC, Enrollment=57/61
        1137 in SURREY by
                Type=LAB, Enrollment=66/191
                Type=LEC, Enrollment=66/192
        1147 in SURREY by Harinder Khangura
                Type=LAB, Enrollment=43/50
                Type=LEC, Enrollment=43/50
        1157 in SURREY by Harinder Khangura
                Type=LAB, Enrollment=52/52
                Type=LEC, Enrollment=52/52
```

Text 2: Output generated by "dumping" the model.

5.4 REST API Hints

- If you have extra getters on your object that you don't want to appear in the JSON, add @JsonIgnore above the method. This is especially useful if the getter returns a reference to another object, which may then reference other objects, …. etc.
- If your object structure differs from what the REST API wants, you can add getter methods to your object to compute/return the necessary data.
  - For example, if your `CourseOffering` object references a `Semester` object, but you need the JSON to list the course offering's year then add a `getYear()` method to `CourseOffering` which delegates getting the year to its contained `Semester` object.

Suggested Development Steps

- Add web UI to your project.
- Suggested order of pages to get working:
  - About
  - Browse (index.html)
  - Graph
  - Watchers
- Use the `curl` commands to exercise each API endpoint independently.
  - Change the ID values in the script to match the values your system produces.
  - Copy and paste the commands, one at a time, to test the end-points.
  - Compare your results to the sample output. Order of values in arrays, and order of fields in output is unimportant (except for the order of data points for the graph).
- Open the developer console in your browser when working with the web UI.
  - For example, F12 in Chrome.
  - Switch to the "Console" tab (or the like) to see error messages.

6. Deliverables

Submit each iteration to CourSys: https://courses.cs.sfu.ca/.
You must create a group in CourSys (even if you worked individually). Submit one solution per group.

6.1 Phase 1 Submission: Dump Model

This is a proof of progress milestone. It shows you have some of the model implemented (parsing data, building a meaningful internal representation). Submit the following:

1. `output_dump.txt`
   - A text file of the complete output generated by `GET /dump-model` via your REST API after it read in and processed the provided input CSV file.
   - Format should be *similar* to Text 2, on page 6.
2. Zip file of project

6.2 Phase 2 Submission: Complete

Full ZIP file of project submitted.

- *No* screen-shots or sample outputs are required.
- Must include the web UI (either as provided by the instructor, on modified by you as needed).

Please remember that all submissions will be compared for unexpected similarities.