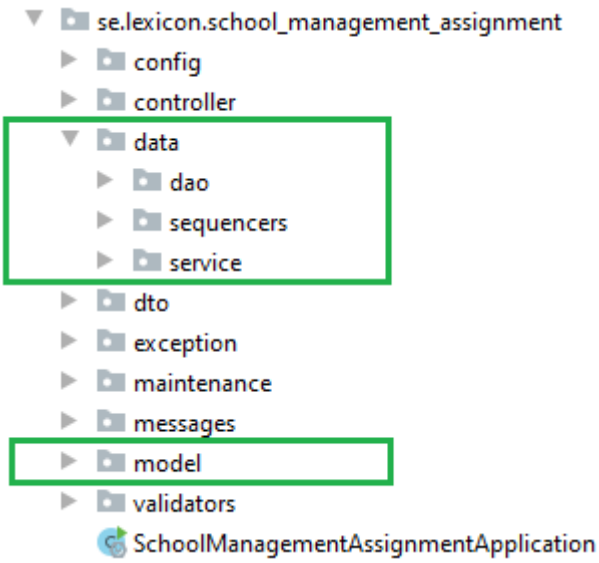
Course Manager

Background:

School Systems Inc. has accepted a contract from a customer that want a Web application written in Java. The task falls on you to create a temporary persistence layer, services and view converters with help of Java Collections. The project is set up and configured and the interfaces are defined. The frontend and web controllers are already created and tested with help of Mocks.
Your task:

1. Create the following:
    a. **`Student.class`** in model package.
    b. **`Course.class`** in model package.
2. Implement the following:
    a. **`StudentDao.class`** interface in **`StudentCollectionRepository.class`**.
    b. **`CourseDao.class`** interface in **`CourseCollectionRepository.class`**.
    c. **`Converters.class`** interface in **`ModelToDto.class`**.
    d. **`StudentService.class`** interface in **`StudentManager.class`**.
    e. **`CourseService.class`** interface in **`CourseManager.class`**.
3. Unit test your implementations and models.



Here you can see the general layout of the system.

The code you are going to create and maintain resides in the **model** and **data** packages.

All **config is already setup** no need to worry about that.

Interfaces with classes are already defined for you. You just write code in the methods.

As for the models you can create them according to the diagram and requirements.

You are not expected to throw any exceptions so remember to check inside of your methods.

If any these following Exceptions are thrown:

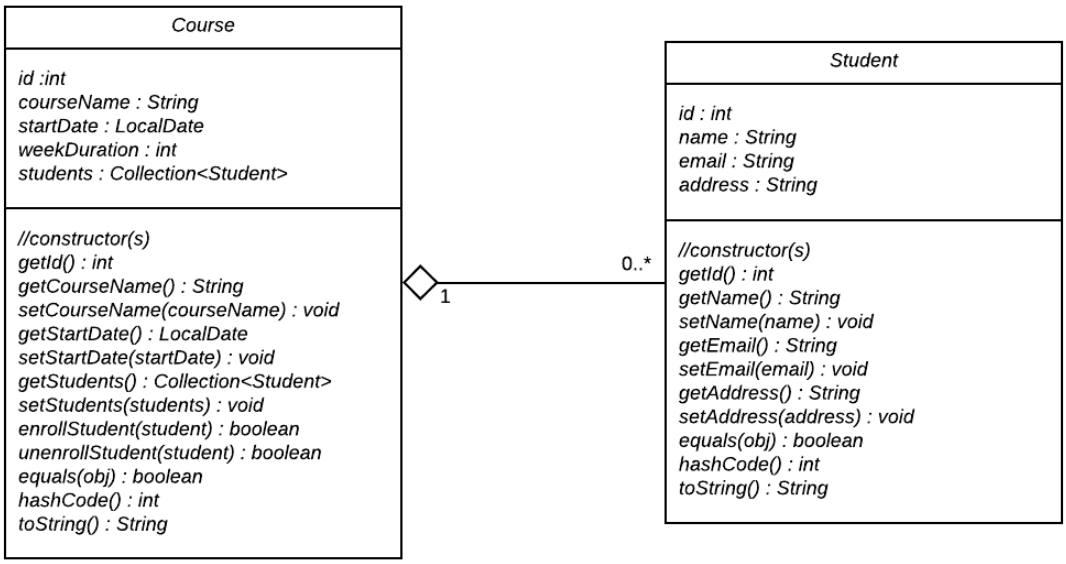
- ResourceNotFoundException
- IllegalArgumentException

These exceptions will be handled by a global exception handler.

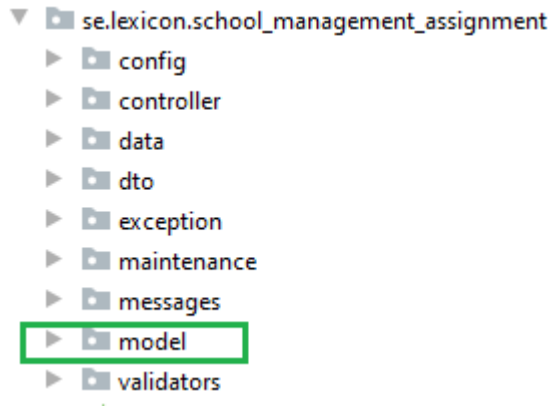
The application is run on an embedded Tomcat Server.

To terminate the server, click 'exit' if you want to save all data you added

Model Requirements:



Model classes **`Student.class`** and **`Course.class`** should be defined in the **model package**:



You need to implement the two models in the model package.

You should also **unit test the models** with standard unit tests in the test scope

The diagram above, defined by a requirements analyst is the summary of our customer wish.

The views and DTO's (Data Transfer Object) are built around this diagram.

No need to null check parameters because validation should be performed in the controller layer automatically.

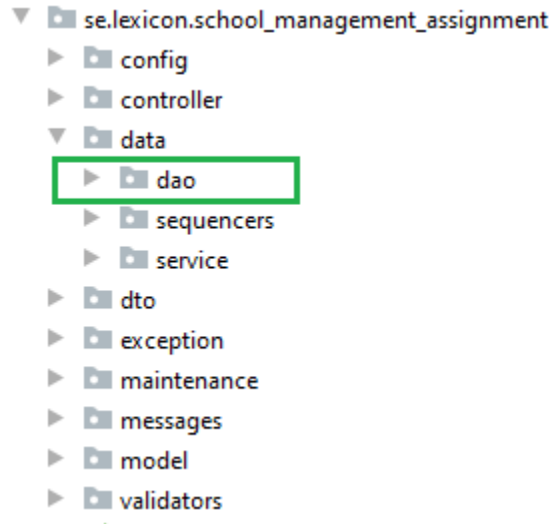
`Student.class`:

- **Fields**:
  - **id** – of type `int` or Integer. Unique attribute, should be set through constructor with **`StudentSequencer.class`** in **`StudentCollectionRepository.class`**
  - **name** – representing the full name of the Student.
  - **email** – unique attribute representing the Student's email.
  - **address** – String representation of the Student's address.
- **Methods**:
  - **Constructor** – minimum requirement is to have an empty constructor **and** one passing in id through constructor.
  - Appropriate **getters / setters according to diagram**
- **@Override**:
  - **equals** – needed internally in Collections
  - **hashCode** – needed internally in Collections
- **Optional @Override**:
  - **toString** for debugging purpose
- **Implement Interface:**
  - **Serializable** – needed in-order to save a Student to a file using Json.

`Course.class`:

- **Fields**:
  - **id** – of type `int` or Integer. Unique attribute, should be set through constructor with **`CourseSequencer.class`** in **`CourseCollectionRepository.class`**
  - **courseName** – representing the name of a Course object (like "Java advanced")
  - **startDate** – of type LocalDate defines the start date of a course object.
  - **weekDuration** – of type int or Integer is a description of the length of a course
  - **students** – representing all objects of Student.class that is enrolled to this course.
- **Methods**:
  - **Constructor** – minimum requirement is to have an empty constructor **and** one passing in id through constructor.
  - Appropriate **getters/setters according to diagram**
  - `public boolean enrollStudent(Student student)` should be used to add a **`Student.class`** object to `Collection<Student> students`. Make sure you avoid adding a duplicate or `null` into the Collection. Should return `true` when student was successfully added, otherwise `false`.
  - `public boolean unrollStudent(Student student)` should be used to remove a **`Student.class`** object from `Collection<Student> students`. Returns `true` when the Student object was successfully removed.
- **@Override**:
  - **equals** – needed internally in Collections
  - **hashCode** – needed internally in Collections
- **Optional @Override:**
  - **toString** for debugging purposes
- **Implement Interface:**
  - **Serializable** – needed in-order to save a course to a file using Json.

DAO requirements:



DAO's (Data Access Object) are responsible to store all objects centrally in applications. In our case it is done with Collections.

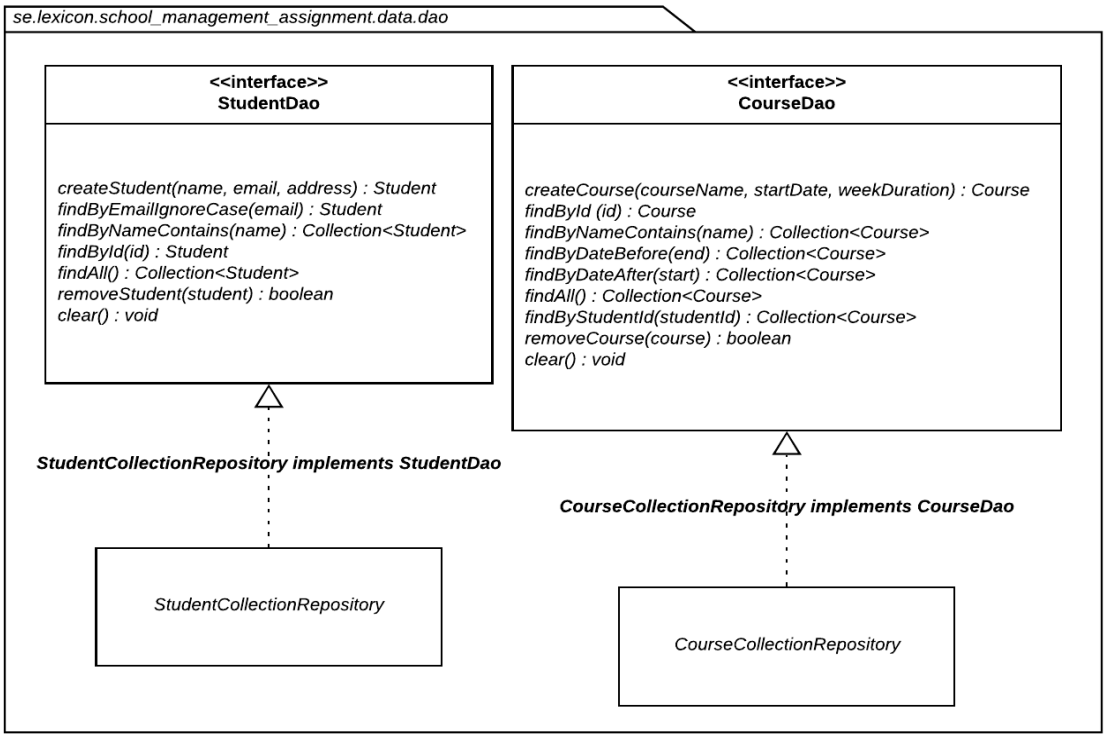
We have two interfaces in the dao package that need proper implementation.

Make sure `CourseCollectionRepository.class` and `StudentCollectionRepository.class` are properly implementing its respective interface.

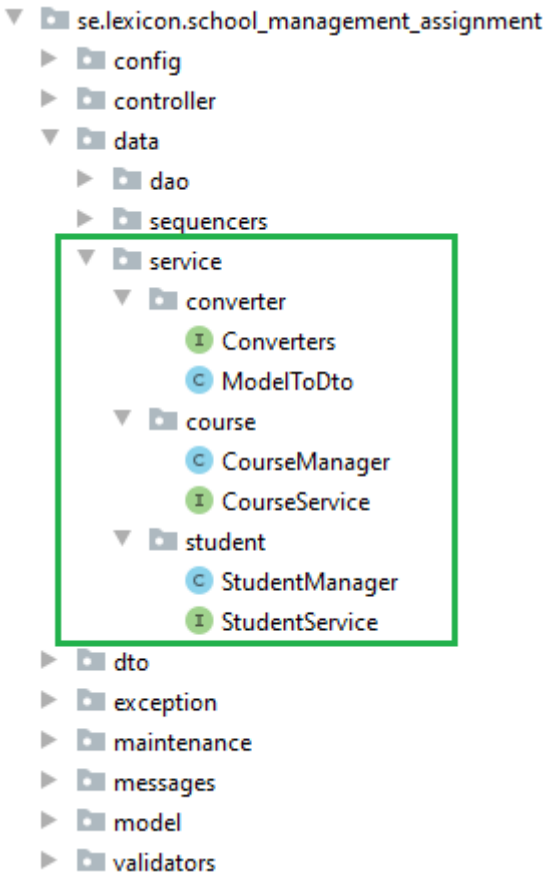
Make sure you use the **sequencers** inside the sequencer package to get id's for the Create methods.

Test classes are already setup for each respective class inside the test scope. Make sure to test the code you write.

Write tests for your methods.



Service requirements:



Service classes are responsible for fetching data from DAO's and sending the appropriate data to the controllers.

In this application we apply the DTO pattern where the purpose is to refine our models into suitable views.

This means traffic from the controllers to the service layer consists of **Data Transfer Objects**. Controllers get **view** objects back in return.

This means you need to **convert** the Student- and Course objects **first** by providing correct implementation to the **Converters** interface in `ModelToDto.class.`

After that you should create the **implementation** of **StudentService** in the `StudentManager.class` and **CourseService** in the `CourseManager.class`

Test classes are already setup for each respective class inside the test scope. Make sure to test the code you write.

You are required to return either an appropriate object or Null from the methods.
Remember to check for Null, can´t do something with nothing.

Write tests for your methods.

Diagram of the service layer: