CMPE 275 Section 1
Spring 2023
Lab 2 - REST, ORM, and Transactions

Last updated: 04/13/2023
Status: Draft

In this lab, you build a set of REST APIs to manage entities and their relationships. Besides REST API, you also get to exercise ORM and transactions. There are two primary types of entities you are concerned with: employees and employers. They have the following relationships and constraints:

- An employee works for *exactly* one employer. Different employees can work for the same employer. Employee IDs must be unique among the same employer's employees.
- Every employee has zero or one manager. An employee's manager, if there is one, must work for the same employer.
- Employees can be collaborators with each other. Collaboration is bidirectional. An employee can have zero or more collaborators, who may or may not belong to the same employer.
- The **name** field is required for every employee.
- The **name** field is required for every employer, cannot be empty or full of white spaces, and needs to be unique as well.

Partial definitions of the related classes are provided below. You must **not** change the type or name of any attribute defined for the given classes. While the Address class is defined for convenience and clarity, you must embed them and **not** store them as separate entities.

Since the employee table has composite primary keys, you want to use JoinColumns for the manager association.

```
package edu.sjsu.cmpe275.lab2;

public class Employee {
    private long id;  // part of the primary key
    Private String employerId; // part of the primary key
    private String name;      // required
    private String email;
    private String title;
    private Address address;
    private Employer employer;
    Private Employee Manager;
    Private List<Employee> reports; // director reports who have the current
employee as their manager.
    private List<Employee> collaborators;

    // constructors, setters, getters, etc.
    …
}

public class Address {
    private String street; // e.g., 100 Main ST
    private String city;
    private String state;
    private String zip;
    …
}

public class Employer {
    private long id;    //primary key
    private String name;// required and must be unique
    private String description;
    private Address address;
    …
}
```

You **must** persist these entities in a relational database, preferably MySQL. You need to create at least three tables, EMPLOYEE, EMPLOYER, and COLLABORATION.

Shallow Form vs Deep (or Full) Form
The shallow form of an entity ignores its nested entities (e.g., reports and collaborators within an employee entity), while a deep form of an entity preserves its nested entities, but all nested entities must use its shallow form.

An example shallow form of an employee entity in JSON :
```
{
  "id": 123,
  “employerId”: “SJSU”,
  "name": "Alice Brown",
  "email": "alicebrown@gmail.com",
  "title": "Test Engineer"
}
```

An example shallow form of an employee entity in XML :

```
<employee>
        <id>123</id>
        <employerId>SJSU</employerId>
        <name>Alice Brown</name>
        <email>alicebrown@gmail.com</email>
        <title>Test Engineer</title>
</employee>
```

An example full form of an employee entity in JSON:

```
{
  "id": 123,
  “employerId”: “SJSU”,
  "name": "Alice Brown",
  "email": "alicebrown@gmail.com",
  "title": "Test Engineer",
  "address": {
    "street": "100 Main ST",
    "city": "San Jose",
    "state": "CA",
    "zip": "95134"
   }
   "employer": {
      "id": “SJSU”,
     “Name”: “San Jose State University”,
     “description”: “Best employer of the world.”,
    },
    "manager": { // shallow form used within the full form
      "id": 100,
      “employerId”: “SJSU”,
      "name": "Carl Dolittle",
      "email": "Carllittle@gmail.com",
      "title": "Test Lead"
    },
    "reports": [  // all in shallow form
           {
                "id": 133,
                 “employerId”: “SJSU”,
                 "name": "Daisy Brown",
                 "email": "daisybrown@gmail.com",
                 "title": "Test Engineer",
           },
           …
     ],
    "collaborators": [ // all in shallow form
           {
                "id": 144,
                “employerId”: “UCSU”,
                 "name": "Edward Smith",
                 "email": "esmith@gmail.com",
                 "title": "Test Engineer",
           }
     ],
}
```

Example of an employer entity’s shallow form:

```
{
      “id”: 572,
      “name”:”The Fine Company”,
      “description”: “Best employer of the world.”
}
```

An example employer entity in XML :

To manage these entities and their relationships, you are asked to provide the following REST APIs. The paths below are relative to the base URL of your app.

For all the API request below, the format parameter format={json | xml } is optional. When it is not present, the output format should be default to JSON. The values of the format are case insensitive; i.e., json and Json are both normalized to JSON.

Employee APIs

(1) Create a employee

Path:
employer/{employerId}/employee?name=XX&email=ZZ&title=UU&street=VV...managerId=WW&format={json | xml }
Method: POST

This API creates an employee object.

- For simplicity, all the employee fields (name, email, street, city, employer, etc), **except** ID and collaborators, are passed in as query parameters. Only the name and employer ID are required. Anything else is optional.
- Collaborators or reports are **not** allowed to be passed in as a parameter.
- If the employee has a manager, only specify the manager's ID using the query parameter *managerId (the employee ID of the manager, who works for the same employer)*. The manager entity must be created already.
- The employer's ID must be specified using the *employerId* path parameter. The employer entity must be created before creating this employee.
- If the request is invalid, e.g., missing required parameters, the HTTP status code should be 400 (if you want to provide detailed error code, e.g., 409 for entity conflicts, that is fine too); if the request is successful, return 200.The request returns the *full form* of the newly created employee entity in the requested format in its HTTP payload, including all attributes.

(2) Get an employee

Path:employer/{employerId}/employee/{id}?format={json | xml }
Method: GET

This returns the full form of the given employee entity with the given employer ID and employee ID in the given format in its HTTP payload.

- All existing fields, including the employer and list of collaborators should be returned. If the employee of the given user ID does not exist, the HTTP return code should be 404; 400 for other errors, or 200 if successful.

(3) Update a employee

Path: employer/{employerId}/employee/{id}?name=XX&email=ZZ&title=UU&street=VV$......
&format={json | xml }

Method: PUT

This API updates an employee object.

- For simplicity, all employee fields (name, email, street, city, etc) that have non-empty value(s), except collaborators, should be passed in as query parameters. Required fields like name must be present. The object constructed from the parameters will completely replace the existing object in the server, except that it does *not* change the employee's list of collaborators.
- Changing an employee's employer is NOT allowed. You can technically achieve this by deleting an entity and creating it with the new employer and employee ID.
- It is not allowed for an employee to report to a manager that does not work for the same employer. Any request that leads to this condition will be rejected with a 400 error.
- Similar to the get method, the request returns the updated employee entity in its full form. If the employee ID does not exist, 404 should be returned. If required parameters are missing or run into other errors, return 400 instead. Otherwise, return 200.
- It is NOT allowed to directly change a person's reports or collaborators.
- Please follow the sample JSON/XML given above.

(4) Delete a employee
URL: http://employer/{employerId}/employee/{id}?format={json | xml }

Method: DELETE

This deletes the employee object with the given ID. No entities returned.

- If the employee with the given employer ID and employee ID does not exist, return 404.
- If the employee still has a report, deletion is not allowed, hence return 400.
- Otherwise, delete the employee within a transaction
    - Remove any reference of this employee from your persistence of collaboration relations
    - If successful, HTTP status code 200 and the deleted employee in the given format, with all values prior to the deletion.

Employer APIs

(5) Create an employer
Path: employer?name=XX&description=YY&street=ZZ&...&format={json | xml }

Method: POST

This API creates an employer object.

- For simplicity, all the fields (name, description, street, city, etc), except ID, are passed in as query parameters. Only name is required.
- The request returns the full form of the newly created employer object in the given format in its HTTP payload, including all attributes. (Please note this *differs* from the generally recommended practice of only returning the ID.)

- If the request is invalid, e.g., missing required parameters, the HTTP status code should be 400 (it's OK to return more detailed error code, e.g., 409 for object conflicts); otherwise 200.

(6) Get a employer

Path:employer/{id}?format={json | xml }
Method: GET

This returns a full employer object with the given ID in the given format.

- All existing fields, including the optional ones should be returned.
- If the employer of the given user ID does not exist, the HTTP return code should be 404, 400 if the request has other errors, otherwise, 200.

(7) Update an employer

Path: employer/{id}?name=XX&description=YY&street=ZZ&...&format={json | xml }

Method: PUT

This API updates an employer object and returns its full form.

- For simplicity, all the fields (name, description, street, city, etc), except ID, are passed in as query parameters. Only name is required.
- Similar to the get method, the request returns the updated employer object, including all attributes in JSON. If the employer ID does not exist, 404 should be returned. If required parameters are missing, return 400 instead. Otherwise, return 200.

(8) Delete an employer

URL: http://employer/{id}?&format={json | xml }
Method: DELETE

This method deletes the employer object with the given ID. No entities returned.

- If there is still any employee belonging to this employer, return 400.
- If the employer with the given ID does not exist, return 404.
- Return HTTP code 200.

Collaboration APIs

(9) Add a collaborator

Path:collaborators/{employerId1}/{employeeId1}/{employerId2}/{employeeId2}&format={json | xml }
Method: PUT

This makes the two employees with the given IDs collaborators with each other. No entities returned.

- If either employee does not exist, return 404, and 400 for any other request error.
- If the two employees are already collaborators, do nothing, just return 200. Otherwise,
- Record this collaboration relation. If all is successful, return HTTP code 200 and any informative text message in the given format in the HTTP payload.

(10) Remove a collaborator

Path:collaborators/{employerId1}/{employeeId1}/{employerId2}/{employeeId2}&format={json | xml }

Method: DELETE

This request removes the collaboration relation between the two employees. No entities returned.

- If either employee does not exist, return 404.
- If the two employees are not collaborators, return 404, or 400 for any other request error. Otherwise,
- Remove this collaboration relation. Return HTTP code 200 and a meaningful text message if all is successful.

Additional Requirements/Constraints

- This is a group assignment for up to *four* team members.
- You must use JPA and persist the user data into a database. For databases, you must use MySQL (or alike, e.g., Amazon RDS), Google App Engine Datastore, Cloud SQL, or Cloud Spanner.
- Please add proper JavaDoc comments.
- You must keep your server running for at least three weeks upon submission. Once your code is submitted to Canvas, you cannot make any further deployment/upload to your app in the server, or it will be considered as late submission or even cheating. You may be asked to show the server log and deployment history upon the TA's request.

Submission

1. Please submit through Canvas a zip file of the whole folder of your source code and resources, including build files. Do not include libs, jars or compiled class files.
2. Please submit a readme.pdf, that includes
    a. The **names and sjsu emails** of your team members
    b. Your **base URL**
    c. *At least 10 sample request queries and screenshot*s, one for each formatted JSON message returned by the ten requests.
3. You must sign up as a group under Groups for Lab 2, and the team **must** submit as a group.

Grading

This lab has a total point of 10, with 9 points for correctness (including persistence, transaction, etc), and 1 point for using the required technologies. You MUST keep your server running in the cloud until the grading is finished.