CMPE 275 Section 1, Spring 2020
Lab 2 - REST and Persistence

Last updated: 03/28/2019 (Released)

In this lab, you build a set of REST APIs to manage the entities and their relationships for a mini gaming website. The API needs to be hosted in some cloud service (e.g., Amazon AWS, Google Cloud, Or Microsoft Azure), and be available live for test and grading.

There are two primary types of entities: Players and Sponsors. They have the following relationships and constraints:

- Opponents: if two players play against each other, they are opponents. The opponent relationship is ***symmetric*** in that is Alice is an opponent of Bob, then Bob is also an opponent of Alice. A player can have zero or more opponent players.
- Sponsors: a player can optionally be sponsored by up to one sponsor. ***Different*** players can have the ***same*** sponsor, and one sponsor can provide sponsorship for multiple players.
- The first name, last name, and email fields are required for every player. Emails have to be ***unique*** across players.
- The name field is ***required*** for every sponsor, and must be unique across all sponsors.
- You must trim the white spaces at the beginning and end of all string attributes and query parameters.

An incomplete definition of the related classes are provided below. You need to embed Address in both Player and Sponsor, and their respective tables, and ***not*** as ***dedicated*** address entities or address tables.

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

public class Player {
   private long id;  // primary key
   private String firstname;
   private String lastname;
   private String email;
   private String description;
   private Address address;
   private Sponsor sponsor;
   private List<Player> opponents;
   // constructors, setters, getters, etc.
  …
}


public class Address {
   private String street;
   private String city;
   private String state;
   private String zip;
   ...
}

public class Sponsor {
   private String name;  // primary key, >= two characters after trimming white spaces
   private String description;
   private Address address;
   private List<Player> beneficiaries;
   ...
}
```

You need to persist entities in a relational database of your own choice. Very likely, you want to create three tables, PLAYERS, SPONSORS, and OPPONENTS.

To manage these entities and their relationships, you are asked to provide the REST APIs defined In the APIs section.

Base URLs

In your readme.txt, you must provide a ***single*** base URL, e.g.,
http://183.57.38.2/
http://183.57.38.2/game-service
https://online-game-service.appspot.com/foo/bar/
The base URL will be used to compose the URL requests to invoke your APIs.

JSON vs XML

By default, your API ***results*** must use the JSON format, unless the query parameter ?format=XML or format=xml is given, in which case you need to use XML.

Shallow Form vs Deep Form

The shallow form of an entity ignores its nested entities (e.g., opponents within a player entity), while a deep form of an entity preserves its nested entities, but all nested entities must use the shallow form.

Sample shallow form in JSON for a player entity:
{

```
  "id": 123,
  "firstname": "Joe",
  "lastname": "Biden",
  "email": "jb@gmail.com",
  "description": "Democratic presidential candidate",
  "address": {
        "street": "1400 Pennsylvania Ave NW",
        "city": "Washington",
        "state": "DC",
        "zip": "20500"
  }
}
```

Sample deep form in JSON for a player entity:

```
{
  "id": 123,
  "firstname": "Joe",
  "lastname": "Biden",
  "email": "jb@gmail.com",
  "description": "Democratic presidential candidate",
  "address": {
        "street": "1400 Pennsylvania Ave NW",
        "city": "Washington",
        "state": "DC",
        "zip": "20500"
  },
  "sponsor": {
        "name": "FreedomAlliance",
        "description": "Some NGO",
        "address": {
              "street": "1300 Pennsylvania Ave NW",
              "city": "Washington",
              "state": "DC",
              "zip": "20500"
        }
  },
  "opponents": [
       {"id": 456,
        "firstname": "Donald",
        "lastname": "Trump",
        "email": "dt@gmail.com",
        "description": "Republican incumbent",
        "address": {
```

```
                    "street": "1500 Pennsylvania Ave NW",
                    "city": "Washington",
                    "state": "DC",
                    "zip": "20500"
          }
     }
 ]
}
```

Sample deep form for a sponsor

```
"sponsor": {
          "name": "FreedomAlliance",
          "description": "Some NGO",
          "address": {
                    "street": "1300 Pennsylvania Ave NW",
                    "city": "Washington",
                    "state": "DC",
                    "zip": "20500"
          },
          "beneficiaries": [
              {"id": 456,
                    "firstname": "Donald",
                    "lastname": "Trump",
                    "email": "dt@gmail.com",
                    "description": "Republican incumbent",
                    "address": {
                              "street": "1500 Pennsylvania Ave NW",
                              "city": "Washington",
                              "state": "DC",
                              "zip": "20500"
          },
            {"id": 123,
                    "firstname": "Joe",
                     "lastname": "Biden",
                    "email": "jb@gmail.com",
                    "description": "Democratic presidential candidate",
                    "address": {
                              "street": "1400 Pennsylvania Ave NW",
                              "city": "Washington",
                              "state": "DC",
                    "zip": "20500"
                }
            }
        ]

    },
```

Player APIs

The resource paths below are relative to the base URL of your app. For all the APIs, return the HTTP status code *200* if the request is successful.

(1) Create a player
Path: player?firstname=XX&lastname=YY&email=ZZ&description=UU&street=VV$...
Method: POST
This API creates a player object.
For simplicity, all the player fields (firstname, lastname, email, street, city, sponsor, etc), except ID and opponents, are passed in as query parameters.
Required Parameters: Only the firstname, lastname, and email are required. Anything else is optional. *Opponents* is **not** allowed to be passed in as a parameter. You are not allowed to provide the ***ID*** of the player either, as the server is supposed to assign the ID.
The ***sponsor*** parameter, if present, must be the ***ID*** of an existing sponsor. The request returns the newly created player object in its ***deep*** form, encoded in the requested format, JSON or XML, in its HTTP payload.
Error Handling: Return the HTTP status code 400 for errors like missing required parameters or bad parameters; return 409 if a player with the same email ID already exists.

(2) Get a player
Path:player/{id}
Method: GET
This returns a deep player object in the requested format in its HTTP payload.

Error Handling: If the player of the given user ID does not exist, the HTTP return code should be 404; if the given ID is not a valid number, return 400.

(3) Update a player
Path: player/{id}?firstname=XX&lastname=YY&email=ZZ&description=UU&street=VV$...
Method: PUT

This API updates a player object.
Same as above, all the player fields (firstname, lastname, email, street, city, sponsor, etc), except opponents, are passed in as query parameters. Required fields like email 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 player's list of opponents.
Opponents is **not** allowed to be passed in as a parameter. The ***sponsor*** parameter, if present, must be the ***ID*** of an existing sponsor. If the sponsor parameter is not provided, it removes any existing sponsorship for this player.
Similar to the get method, the request returns the updated player object in its deep form.

Error Handling: If the player entity does not exist for the given valid ID, 404 should be returned. If required parameters are missing, return 400 instead. Otherwise, return 200.

(4) Delete a player
URL: http://player/{id}
Method: DELETE

This deletes the player object with the given ID, and returns the deep form of the deleted player. When a player is deleted, the relevant opponent and sponsoring relationships are ***cascadingly*** removed too.
Error Handling: If the player with the given ID does not exist, return 404. Return 400 for other bad requests.

Sponsor APIs

(5) Create an sponsor
Path: sponsor?name=XX&description=YY&street=ZZ&...
Method: POST

This API creates a sponsor object.
For simplicity, all the fields (name, description, street, city, etc) are passed in as query parameters. Only name is required.
The beneficiaries ***cannot*** be passed in as a parameter.
The request returns the newly created sponsor object in its deep form in the requested format in the HTTP payload.
Error Handling: If the sponsor object already exists, return 409. For other bad requests, return 400.

(6) Get a sponsor
Path:sponsor/{name}
Method: GET

This returns a deep sponsor object in the requested format in its HTTP payload.
Error Handling: If the sponsor of the given name does not exist, the HTTP return code should be 404; otherwise, 200.

(7) Update a sponsor
Path: sponsor/{name}?description=YY&street=ZZ&...
Method: PUT

This API updates a sponsor object.
For simplicity, all the fields (description, street, city, etc), except name, are passed in as query parameters. Only name is required.
The beneficiaries ***cannot*** be passed in as a parameter.
Similar to the get method, the request returns the updated sponsor object in its deep form, including the ***shallow*** beneficiaries.

Error Handling: If the sponsor name does not exist, 404 should be returned. If required parameters are missing, return 400 instead. Otherwise, return 200.

(8) Delete a sponsor
URL: http://sponsor/{name}
Method: DELETE

This method deletes the sponsor object with the given name, and returns the deep form of the deleted sponsor object.

Error Handling: If there is still any player benefiting from this sponsor, return 400. If the sponsor with the given name does not exist, return 404.

Opponent's APIs

(9) Add an opponent relationship
Path:opponents/{id1}/{id2}
Method: POST

This makes the two players with the given IDs opponents with each other.
If the two players are already opponents, do nothing, just return 200. Otherwise, record this opponent relationship.
Error Handling: If either player does not exist, return 404.
Return 400 for other bad requests, e.g., the given two players are the same.
If all is successful, return HTTP code 200 and any informative text message in the HTTP payload.

(10) Remove an opponent relationship
Path:opponents/{id1}/{id2}
Method: DELETE

This request removes the opponent relation between the two players.
Error Handling: If either player does not exist, return 404. Return 400 for other bad requests; e.g, if the two players are not opponents, return 400 too. Otherwise,
Remove this opponent 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 relational database. You can choose one of the following relationship databases, MySQL (preferred), Amazon RDS, Google Cloud SQL, or Azure SQL Database. .

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 screenshots, 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 points of 10, all based on correctness. If, however, you are not using JPA or not meeting any other technical requirement (e.g., relational database), you will get at least 50% penalty.

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