CMPE 275 Section 1, Fall 2020

Lab 2 - REST and Persistence

**Last updated: 11/22/2020 (Published)**

In this lab, you build a set of REST APIs to manage entities and relationships in a mini gaming website (you can give your own fancy name for it). The API needs to be hosted in cloud services like Amazon EC2, Google App Engine, Or Compute Engine. You must use JPA for persistence, and each API method must be transactional.

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 if 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 an external sponsor. ***Different*** players can have the ***same*** sponsor.
* The first name, last name, and email fields are required for any player. Emails have to be ***unique*** across players.
* The name field is ***required*** for any Sponsor, and does ***not*** need to be unique.

Partial definition of the related classes are provided below. While the Address class is defined for convenience and clarity, you are recommended to embed addresses and not to store them as separate entities.

package edu.sjsu.cmpe275.lab2;

public class Player {

    private long id;

    private String firstname;

    private String lastname;

    private String email;

    private String description;

    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 long id;

    private String name;

    private String description;

    private Address address;

    private List<Player> players;

    ...

}

You need to persist these entities in a non-volatile database of your own choice. You may want to create three tables, PLAYER, SPONSOR, and OPPONENTS.

**Shallow Form vs Deep (or Full) 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.

An example shallow form in JSON for a player entity:

{

  "id": 123,

  "firstname": "Joe",

  "lastname": "Biden",

  "email": "jb@gmail.com",

  "description": "Democratic presidential candidate"

}

An example deep form in JSON for a player entity:

{

  "id": 123,

  "firstname": "Joe",

  "lastname": "Biden",

  "email": "jb@gmail.com",

  "description": "Democratic presidential candidate",

  "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",

       }

 ]

}

An example deep form for a sponsor

"sponsor": {

"name": "FreedomAlliance",

"description": "Some NGO",

"address": {

     "street": "1300 Pennsylvania Ave NW",

   "city": "Washington",

   "state": "DC",

   "zip": "20500"

},

"players": [

          {"id": 456,

   "firstname": "Donald",

   "lastname": "Trump",

   "email": "dt@gmail.com",

   "description": "Republican incumbent",

   },

        {"id": 123,

"firstname": "Joe",

  "lastname": "Biden",

   "email": "jb@gmail.com",

   "description": "Democratic presidential candidate",

   }

 ]

  },

In order to manage these entities and their relationships, you must provide the following REST APIs. The paths below are relative to the base URL of your app. Every API result contains an HTTP status code (200, 400, etc) and a payload (e.g., an JSON object).

*Player APIs*

**(1) Create a player**

**Path: player?firstname=XX&lastname=YY&email=ZZ&description=UU&...**

**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. Only the firstname, lastname, and email are required. Anything else is optional. Opponents are not allowed to be passed in as a parameter.

The sponsor parameter, if present, must be the ID of an existing sponsor. The request returns the deep form of the newly created player object in JSON in its HTTP payload, including all attributes. (Please note this differs from 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; otherwise 200.

**(2) Get a player**

**Path:player/{id}**

**Method: GET**

This returns a deep player object in JSON in its HTTP payload.

All existing fields, including the optional sponsor and list of opponents should be returned.

The JSON should contain the full sponsor object, if present.

The list of opponents can be either (a) list of player IDs, or (b) list of “shallow” player objects that do not have their opponents list populated. If you take option (b), you want to use techniques like lazy loading to avoid serializing the whole game network starting from the requested player in the JSON to be returned.

If the player of the given user ID does not exist, the HTTP return code should be 404; otherwise, 200.

**(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, and returns the full player object.

For simplicity, all the player fields (firstname, lastname, email, street, city, sponsor, etc), except opponents, should be 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.

Similar to the get method, the request returns the updated player object, including all attributes (first name, last name, email, opponents, sponsor, etc), in JSON. If the player ID does not exist, 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.

If the player with the given ID does not exist, return 404.

Otherwise, delete the player and remove any reference of this player from your persistence of opponent relations, and return HTTP status code 200 and the deep form of the deleted player in JSON.

*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), except ID, are passed in as query parameters. Only name is required.

The request returns the deep form of the newly created sponsor object in JSON in its HTTP payload, including all attributes. (Please note this differs from 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; otherwise 200.

**(6) Get a sponsor**

**Path:sponsor/{id}**

**Method: GET**

This returns a full sponsor object in JSON in its HTTP payload.

All existing fields, including name, description, street, and city, should be returned.

If the sponsor of the given ID does not exist, the HTTP return code should be 404; otherwise, 200.

**(7) Update a sponsor**

**Path: sponsor/{id}?name=XX&description=YY&street=ZZ&...**

**Method: PUT**

This API updates a sponsor object.

For simplicity, all the fields (name, description, street, city, etc), except ID, are passed in as query parameters. Only the name is required. The existing sponsor will be fully updated with the given attributes, with missing attributes removed.

Similar to the get method, the request returns the deep form of the updated sponsor object, including all attributes in JSON. If the sponsor ID does not exist, 404 should be returned. If required parameters are missing, return with error code 400 instead. Otherwise, return 200.

**(8) Delete a sponsor**

**URL: http://sponsor/{id}**

**Method: DELETE**

This method deletes the sponsor object with the given ID.

If the sponsor with the given ID does not exist, return 404.

If there are players who are sponsored by this sponsor, the sponsorship is automatically removed.

Return HTTP code 200 and the deleted object in its full form in JSON if the object is deleted;

*Opponent’s APIs*

**(9) Add an opponent**

**Path:opponents/{id1}/{id2}**

**Method: PUT**

This makes the two players with the given IDs opponents with each other.

If either player does not exist, return with error code 404.

If the two players are already opponents, do nothing, just return 400.

Otherwise, record this opponent relationship. If all is successful, return HTTP code 200 and any informative text message in the HTTP payload.

**(10) Remove an opponent**

**Path:opponents/{id1}/{id2}**

**Method: DELETE**

This request removes the opponent relation between the two players.

If either player does not exist, return with error code 404.

If the two players are not opponents, return with error code 400. 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 database. For databases, you must use MySQL, 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
   1. The **names and sjsu emails** of your team members
   2. Your **base URL**
   3. 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 point of 10. You MUST keep your API running in the cloud until the grading is finished.