SER421 Fall 2021 Lab5 **(REST) API-driven development**

Objectives:
1. Gain proficiency in best practices for consuming APIs.
2. Gain competency in using AJAX and/or fetch.
3. Implement a REST API.

Overview:
For this lab you will construct your own REST API and consume it.
NOTE: Extra credit is offered for doing test-driven development of your API. Review the specification of how this is done BEFORE you start coding your API or you will not be able to receive the extra credit!

Constraints on all parts of the application:
1. Unlike other labs, there are no limitations on external libraries you may use on the server or the client. On the server, you may use Node HTTP, Express, or REST libraries like restify. On the client, you may use any frameworks, CSS or libraries you like.
2. You are required to implement robust error handling. By this I mean you must catch all 4xx and 5xx errors.
3. Your submission must be in a zipfile named lab5.zip. For this lab you are free to package your solution as you like, such as just submitting one more .js files and telling us what to npm install and how to run, or by giving us a complete and correct package.json (if you do not know what this is do not do it!).
4. IMPORTANT: Do ***not*** upload your *node_modules* subdirectory. This is where npm will place all of your 3rd party libraries for express and its dependencies. This will create very large file uploads that could fail. Instead, tell us what to npm install or give us the package.json. As always, a complete Readme is how you tell us!

Task: Make a golf app!
2 parts: creating a client-side API-driven app for your Lab 1 Task 2 golf console program, and creating an API to serve the content required for your golf app. You can start with your Lab 1 Task 2 solution, or you may use our solution.

Part 1: Make a REST API of the golf domain (60 points)
In your lab 1 task 2 you wrote a golf program that had a Tournament with Players in the Tournament. You will use this domain, and your Lab 1 Task 2 code (or ours), to create a REST API. Domain constraints:

1. Your API should support many Tournaments, not just one.
2. Tournaments are uniquely identified by the *name* attribute. The remaining attributes are exactly the same as in the json given in Lab 1.
3. Tournaments have zero or more Players.
4. Tournaments are immutable once the Tournament is completed.
5. A Player may only be participating in one Tournament at a time. A Player does not have to be participating in a Tournament at any given time. A Player not currently participating in a Tournament has no score or hole.
6. A Player is uniquely identified by the concatenation of her/his lastname and first initial.

API specification: provide a RESTful API endpoint for the following
1. Create a Tournament from JSON. The Tournament may be initialized with Players if there are players in the JSON (as given in Lab 1), but a Tournament may also be initialized with no Players.
2. Add a Player to a Tournament.
3. Remove a Player from a Tournament.
4. Allow modifications to the *award* and *yardage* attributes of a Tournament
5. Retrieve all Tournaments. The resulting value should not include the Players in the Tournament, just the Tournament metadata (attributes).
6. Retrieve all Players in a given Tournament
7. Retrieve all Tournaments in a given year range, i.e. lb < YEAR < ub.
8. Retrieve all Players in a Tournament with a score less than or equal to a given parameter, i.e. score < param
9. Create a Player (the new Player will not be in any Tournament)
10. Delete a Player only if that Player is not in any Tournament
11. Update the Player's score if the Player is in a Tournament (see Lab 1 "postScore()")
12. Retrieve all Players not in any Tournament.

Your API server may accept Content-Type form-data for any input except where indicated (#1). Your return payloads should be in JSON. Each of the 12 endpoints above is worth 5 points. 1 point is simply getting it working, no matter the RESTfulness of the design. 1 point is for proper error-handling with respect to the endpoint. 1 point for providing API documentation of each endpoint in your README or in an online format - HTML, or using a documentation tool like apidoc (there are lots of tools you can try). The remaining 2 points are for proper REST design – using the correct verbs, response codes, and REST properties we will discuss in class the week of 11/15.

Part 2: Construct a client-side app using your API (40 points)

Write a client-side app that exercises your API. We do not provide a sample UI for this lab task, but your app should demonstrate the following features under the given constraints:

1. Your landing page should list the set of available Tournaments with their metadata but not the Players in the Tournament (API5). The list should be filterable by year range (API7).
   There should be an ability to expand each Tournament (you decide how the UI implements *expand*) to:
   a. show the Players in the Tournament (API6), and filter that display by Players whose score is less than or equal to a given value (API8)
   b. add a Player to the Tournament (API2)
   c. remove a Player from a Tournament (API3)
   d. modify the award and/or yardage attributes of the Tournament (API4)
   e. Update a Player's score for the next hole (API11)
   f. Indicate if the Tournament is completed, and if so who the winner is and the score (which API here?)
2. This page should also include the ability to create a new Tournament from JSON (API1). You may design how you want this to appear in the UI – as a separate page or not, and whether to use a textarea to cut-and-paste the JSON in (as in Lab 4), or to use a file upload feature (as given in your course examples).
3. Your app should have features to list all Players not in a Tournament (API12), to add a Player (but not to any Tournament, API9), delete a Player not in a Tournament (API10)
4. Your app should include a feature to display a Tournament leaderboard, and a feature to display the projected leaderboard using the "projectScoreByIndividual" and "projectScoreByHole" features from Lab 1. Note how these are not part of your API – the leaderboard and projected leaderboard should be implemented entirely on the client side!
5. The UI should never show a stacktrace for an error, but rather display a user-friendly message which include the response code and a human-readable message pinpointing the error.
6. All API calls must be done using AJAX or fetch.

Extra Credit (24 points)

The extra credit, as promised, is demonstrating a Test-driven development (TDD) process as part of completing your API (Part 1). To do this, you will need to create a test first, have it fail, then write the code to make it succeed. You will demonstrate this by using git. You will make a local repo and commit your test first, then commit your code that makes the test succeed. In your commit comment you will indicate what test it is (if committing a test) or what should now succeed (if you are writing the code to make the test pass). You may use the API number above to identify each commit. You must do one test and one resolution of a failed test at a time (in other words, do not write all the tests and then all the code, work one at a time. This is called "red-green-refactor"). Note that you do not have to put your code on GitHub to do this (if you choose to sync it with GitHub, then make sure your repo is private, and add kgary and L1nu5 [your TA Abhishek] as collaborators). To earn this extra credit, you must commit every single step of your work, so there should be a lot of commits! Test code is also required, you cannot merely use Postman or Arc or cUrl to construct these tests.

Strictly speaking, testing an API is not really a *unit test* but more of an *integration test*, but no matter. You will write a test of each of the 12 API endpoints and receive 2 points for each TDD-completed test. You have a lot of leeway in how you decide to test. For example, you can write custom http client code in NodeJS (there is an example in your NodeHTTP directory) or use a framework like Mocha, Jest, or supertest (I like the last one for this and there are some blogs on it). No matter what you use you must test the *happy day case* (works as expected, response code 2xx) and client error cases (4xx response codes). You do not have to worry about 5xx cases. Each test should check for the proper response code, other expected response headers, and any expected payload. Your tests should use NodeJS assertions (https://nodejs.org/api/assert.html or https://www.w3schools.com/nodejs/ref_assert.asp), or some assertion mechanism that clearly indicates failure/success (some test frameworks may have a built-in mechanism).

Finally, yes I am OK if you start TDD but then decide to abandon it if you feel it too time-consuming (it really isn't once you get the hang of it, as Derek said it is been shown to be a productivity booster). I will give partial credit, 2 points per completed red-green-refactor cycle (test and resulting code to make it pass).