AngularJS Triple Lab

Overview for Triple Lab Parts 1-3:

In previous labs, you created a database with two tables in it (your "User" table and your "Other" table). You then created a Web API (JSP page) that retrieved records from your "Other" database table and displayed them in JSON format. Over the next three labs, you will create these additional Web APIs (JSP pages):

- 4 for your "Other" table: get all records (copy/paste from old lab), get one record, insert, update, and delete.
- 2 for your "User" table: get all records, insert (register).
- 1 for logon and 1 for logoff.

You will create a user interface (using HTML, CSS, and JavaScript with help from the AngularJS JavaScript framework) that enables the user to register, logon/logoff, see a list of users, and see a list of the "other" records. If (and only if) the user is logged on, the user shall be able to insert/update/delete "Other" records. Your client code shall utilize AngularJS routing with multiple small JS controller files as presented in sample code.

Note: we will be implementing server side validation even though client side validation could be employed. Why? Server side code needs to be written robustly and it must provide good error reporting. It is very easy to add client side validation (like you did for the form in responsive design lab 2), but we cannot rely solely on client side validation because that would lead to a very "leaky" and inconsistent approach to validation. I will demonstrate in class (or maybe a lab activity) how easy it is to add client side validation (after the fact) to prevent server side validation errors from occurring and to provide a more responsive user validation experience.

This "triple lab" will be submitted in three parts, one part per week. Each submission is cumulative: Part II requirements include Part I requirements; Part III requirements includes the requirements of Parts I and II.

Database Requirements

Double check that your database design meets the requirements of the database lab.

- If your database design does not meet the requirements (for example, if it does not have a non-required non-character), you will lose points in many areas from your Triple Lab grade.
- If you have used any SQL keywords as table or field names, it is very likely that your SQL commands (SELECT, INSERT, UPDATE, DELETE), as issued by your Java code, will not work properly.
- If you need to modify your database design, do so now (so you won't receive large deductions for this triple lab) and then make sure that your Web API lab still works (so you won't get deductions on your project grade for that functionality).
- The database lab also had requirements for your data. For example, you were to have at least a certain number of realistic looking records in each table and your “other” table needed to contain valid image URLs. As you work through your Triple Lab, you will certainly be inserting, updating, and deleting records, but try to maintain the minimum number of realistic looking records at all times. You can do this, for example, by using MySqlWorkbench if you need to quickly insert a bunch of test records so that you can debug delete functionality.

Web API: API stands for Application Programming Interface (essentially a method call that has no user interface, a method that is exposed so that it can be called by external code). A Web API then is a server generated page (e.g., JSP page or PHP page or aspx page) that is only data (e.g., JSON format), has no HTML user interface.
Project Organization / Design Specifications

You will be producing a lot of files as you complete your Triple Lab and it is important to organize them properly. Plan to organize your files as shown below. This is not a complete list of all the files you’ll need, but it is probably complete for CRUD (Create-insert, Read-select, Update, and Delete) for one table (Person, in my example).

Note that my example differs a bit from your requirements. If CountryFlag would be my “other” table and Person would be my “user” table, I should have Web APIs for getting all flags, for deleting a flag, for inserting a flag, for getting a particular flag, for updating a flag, for inserting a person, for getting all persons, as well as one for log on and one for log off.

All of your work for the Angular Triple lab shall be stored in a folder (under your web root folder) named "tripleLab".

tripleLab/htmlPartials shall contain templates that will be pulled into tripleLab/index.html (based on routing rules), similar to the sample code in "07_routing_for_layout".

Your javaScript files shall be organized by controller, in tripleLab/js (all of these js files shall be involved by tripleLab/index.html).

One of the js files shall specify routing rules.

tripleLab/webAPIS shall store all of your web APIs (JSP pages). These files shall reference the java classes in "Source Packages".

tripleLab/index.html shall be the main page (of your Single Page Application) for your Triple Lab.

These are the java packages and classes that are referenced by your Web APIs (JSP pages).

Have one model.XXX for each database table (where XXX is your database table name).

Right click "Libraries" and add the mysql database driver JAR file and the gson jar file that handles JSON ↔ POJO conversion (POJO means Plain Old Java Object).
Part I Server Side Web API Requirements (JSP Pages):

1. **getOtherListAPI.jsp** *(copy/paste your webAPI.jsp from a previous lab)* shall return a JSON string that contains:
   - if no database error:
     - a database error message containing ""
     - an array of objects (one object per row of data in your Other table) where each object has all the columns in your Other table (plus a record level error message). *If any record has null in its non-character null-able field, convert this to "" (empty string) so you don't throw any null pointer exceptions.*
   - otherwise (database error):
     - the database error message
     - an empty array, e.g. [ ], no objects in the array.

   Note: one easy way to test the database error case is to not be tunneled in when testing from home.

2. **getUserListAPI.jsp** *(similar to getOtherListAPI)* shall return a JSON string of an object with one of the following:
   - if no database error:
     - a database error message containing ""
     - an array of objects (one object per row of data in your User table) where each object has all the columns in your User table (plus a record level error message).
   - otherwise (database error):
     - the database error message
     - an empty array, e.g. [ ], no objects in the array.

3. **insertUserAPI.jsp** shall:
   - Accept a (JSON-ized) "StringData" object (which has one String property per column in your User table plus a record level error property). This object represents the User record that is to be inserted into the database.
   - Validate (server side validation) all the values from the JSON string and, if all are valid, attempts to insert the data as a record in your User table.
   - Return back to the page a (JSON-ized) StringData object (with each field holding the possible field level error messages) and a record level error message.

   - If the insert was successful, then all the properties of the StringData object shall contain "" (empty string), signifying that all fields passed validation. The record level message shall also contain "".
   - Otherwise, the StringData properties shall contain field level error messages (see examples below) and a record level error message.

**Examples of field level error messages:** "Required", "Please enter an integer", "Please enter a number", "Please enter a valid date", "Exceeds maximum (xx characters)" (for this case, and show them the first xx characters that they entered and allow them to modify their input to fit within the allowable number of characters).

**Examples of record level error messages:** the database exception message (as provided by MySql in your Java catch block), or "database unavailable", "that email address already exists", or "you must be logged in to perform this operation" (you will implement this error message in Part III). To be clear, you are being asked to "intercept" known errors (like the tunneling error or duplicate value for unique field) and convert them to user friendly messages, but for other errors (unknown/unexpected error like sql syntax error), just show the actual database exception message.
Part I Client Side Requirements (HTML, CSS, JavaScript):

1. Your tripleLab/index.html shall be a Single Page Application (or SPA, which is similar to Single Page Design, but not the same) that manages its layout using the “ng-view” directive as a placeholder for your content (like Angular sample code "07_routing_for_layout").
   - Your tripleLab/index.html shall NOT have any ng-include statements (like you did in lab 1 where you included top.html and bottom.html).
   - Your layout can be simple, but try to make the color scheme similar to your home page. If you decide to use background images, make sure that they do not interfere with the readability of your content.

2. The nav bar of tripleLab/index.html shall have 4 working links ("Home", "Register", "Users", and "Labs") and a few links that will be made to work in a future lab.
   - The Home Link shall (using a js file as controller) route to an HTML partial page that calls your "Get Other List Web API" and then displays all that data (all row, all columns) in a HTML table that’s in a HTML partial page (like Angular example "05_get_with_repeat"). The URL data field shall become a src attribute of an <img> tag so that images (not URLs) are shown on the page. Each row of data shall have a view icon, an update icon, and a delete icon, similar to the image below (the icons will be made to work in future labs). The HTML table shall provide Sort/Filter functionality (when the user clicks on the column heading) like Angular example "06_sorting_and_filtering".

   ![Flag List](image)

   - The Users Link shall (using a js file as controller) route to an HTML partial page that:
     - calls GetUserList Web API and displays the results (all rows, all columns of data) in an HTML table in an HTML partial page using ng-repeat. Sorting/filtering is optional. This page shall not have update/delete/view icons for each data row.
   - The Register Link shall (using a js file as controller) route to an HTML partial page that implements “insert user” by calling the Insert User Web API. Once the Register link has been pressed, the user shall see labeled textboxes (one for every column of data in your user table) and a "register" button. The password textbox shall be an <input> tag with type="password" (not type="text") so that the characters show up like dots instead of visible characters. When the register button is clicked, the Insert User Web API shall be invoked (passing a JSON-ized version of a user object with all the user inputs as attributes). The functioning of the API was discussed earlier in this document. When the page receives the AJAX response, the page shall display all field level error messages (which would each be “” empty string if the field passes validation), as well as a record level error message (e.g., success or a DB error or a validation message like "please try again").
   - The nav bar shall have these links (that will be made to work in a future lab): Log On, and Log Off.
   - The Labs Link shall link out of the triple lab SPA, to your labs page: `<a href="../labs.html">Labs</a>`

3. Blog in your Labs page that discusses your work, provides a link to tripleLab/index.html and direct links to each of the three Web APIs that you wrote this week (each link having an example URL that shows it’s working).
Part II Web API (Server Side) Requirements

1. insertOtherAPI.jsp shall:
   o Accept a (JSON-ized) "StringData" object (which has one String property per column in your Other table plus a record level error property). This object represents the Other record that the user wishes to insert.
   o Validate (server side validation) all the values from the JSON string and, if all are valid, attempts to insert the data as a record in your Other table. *Remember that "" (empty string) is a valid entry for your non-character null-able field. If the user types "" for this field, you encode null into the database when executing the INSERT statement. Your sample code is already designed to handle this – see methods like setDate, setInt, setBigDecimal in class dbUtils.PrepStatement.*
   o Return back to the page a (JSON-ized) StringData object (with each field holding the possible field level error messages) and a record level error message.
     ▪ If the insert was successful, then all the properties of the StringData object shall contain "" (empty string), signifying that all fields passed validation. The record level message shall also contain "".
     ▪ Otherwise, the StringData properties shall contain field level error messages and a record level error message – see example error messages below.

2. updateOtherAPI.jsp shall:
   o Accept a (JSON-ized) "StringData" object (which has one String property per column in your Other table plus a record level error property). This object represents the Other record that the user wishes to insert into the Other table. For update, the id field must be populated (it was not necessary for insert).
   o Validate (server side validation) all the values from the JSON string and, if all are valid, attempts to insert the data as a record in your Other table. *Remember that "" (empty string) is a valid entry for your non-character null-able field. If the user types "" for this field, you encode null into the database when executing the INSERT statement. Your sample code is already designed to handle this – see methods like setDate, setInt, setBigDecimal in class dbUtils.PrepStatement.*
   o Return back to the page a (JSON-ized) StringData object (with each field holding the possible field level error messages) and a record level error message.
     ▪ If the insert was successful, then all the properties of the StringData object shall contain "" (empty string), signifying that all fields passed validation. The record level message shall also contain "".
     ▪ Otherwise, the StringData properties shall contain field level error messages and a record level error message – see example error messages below.

Examples of field level error messages: “Required”, “Please enter an integer”, “Please enter a number”, “Please enter a valid date”, “Exceeds maximum (xx characters)” (showing them the first xx characters that could fit).

Examples of record level error messages: the database exception message (as provided by MySql in your Java catch block), or "database unavailable", “that product name already exists”, or "you must be logged in to perform this operation" (you will implement this error message in Part III). To be clear, you are being asked to “intercept" known errors (like the tunneling error or duplicate value for unique field) and convert them to user friendly messages, but for other errors (unknown/unexpected error like sql syntax error), just show the actual database exception message.

3. getOtherAPI.jsp shall return a JSON string of an object:
   o A StringData object (from your "Other" table). If there is no database error, the its record level error message shall contain "". Otherwise, it shall contain the database error message. *If the record has null in its non-character null-able field, convert this to "" (empty string) so you don’t throw any null pointer exceptions.*
4. **deleteOtherAPI.jsp** shall:
   - Accept an id which indicates the record from the Other table that the user wishes to delete.
   - Attempt to delete the record from the Other table (the record with the given id)
   - Return an error message (e.g., "database is unavailable" or "that record never existed" or "you must be logged in to perform this operation") OR empty string "" if the delete was successful.

**Part II Client Side Requirements (HTML, JavaScript)**

1. **Insert Other – Add an “Insert” link to the Home page** *(the page that shows all of your “Other” data).*
   - When the user clicks on Insert button from the "Other Data List", the user interface shall display an "Insert Other Page" which has a prompt, textbox, and error message for each field of your "Other" database table plus an "Insert Save" button, plus a "back to list" button.
   - When the user clicks the Insert Save button, all of the user's data shall be converted to JSON and sent to the Insert Other Web API. When the page receives the response from the API, the page shall display all of the field level error messages plus the record level error message. If the insert was successful, it shall display a success message. *Remember to test inserting a record with null for your null-able non-character field.*
   - When the user clicks the "back to list" button, they are back at the main Other Data List UI.

2. **Update Other.**
   - When the user clicks on an Update icon from one of the rows of the "Other Data List", the user interface shall display an "Update Other Page". The Update Other Page shall have a prompt, a textbox, and an error message for every field of your "Other" database table.
   - All the textboxes shall be pre-filled with data pulled from the record of the update icon which the user initially clicked. *Remember to test updating a record that has null for your null-able non-character field.*
   - The page shall also display an "Update Save" button which, when clicked, sends all of the user's data (converted to JSON) to the Update Other Web API. When the page receives the response from the API, the page shall display all of the field level error messages plus the record level error message. If the update was successful, it shall display a success message. *Remember to test updating a record so that it has null for your null-able non-character field.*
   - This page shall have a "back to list" button. When the user clicks the "back to list" button, they are back at the main Other Data List UI.

3. **Delete Other.** When the user clicks on the delete icon of a row, the page shall invoke the Delete Other Web API (passing in the id of the record to be deleted). When the page receives the response from the API, it shall refresh the data that is listed on the page.

4. **View Other (optional).** When the user clicks on the view icon of a row, the page shall invoke the Get Other Web API (passing in the id of the record to be viewed). When the page receives the response from the API, it shall label and display all the fields of the record on the page (replacing the data list). This page shall have a "back to list" button.

5. **Blog** in your Labs page that discusses your work, provides a link to tripleLab/index.html and direct links to each of the four Web APIs that you wrote this week (each link having an example URL that shows it's working).
Part III Web API (Server Side) Requirements

1. **logonAPI.jsp** shall:
   - Accept a username and password - extract from URL using request.getParameter().
   - If logon was successful, write the username into the JSP session object (which can be checked by the other APIs) and return a StringData object (JSON-ized) fully populated with the user’s record (from the user table).
     ```java
     session.setAttribute("loggedInUser", foundUserName);           // if credentials are in DB
     ```
   - If logon is unsuccessful, invalidate the JSP session object then return an error message such as "invalid credentials", "database is unavailable" or (while debugging) whatever SQL syntax error message may have been generated by a database exception.
     ```java
     session.invalidate(); // if credentials not found in DB
     ```

   Other database modification APIs shall check the JSP session object to be sure the user is logged in before they allow the database modification operation. Remember that you must use the same name (e.g., "loggedInUser") when you extract from the session as you used when you originally put information into the session (in a different JSP page). Also, session.getAttribute() returns a plain java "Object" so you have to type cast it to the correct type before you can use it.
   ```java
   String loggedOnUserName = (String) session.getAttribute("loggedInUser");
   ```

2. **logoffAPI.jsp** shall require no input. It shall invalidate the JSP session object and then send a confirmation message back to the html page. This is how you invalidate the session in the JSP page:
   ```java
   session.invalidate();
   ```

Part III Client Side Requirements (HTML, JavaScript)

1. **Logon** shall provide an email address textbox, a password textbox, and "log in" button. The password textbox shall be an `<input>` tag with type="password" (not type="text") – this will make it so characters show up like dots in the password field. When the "log in" button is clicked, the Logon Web API (previously described) shall be invoked (passing in the email address and password). When the response is received from the API, the page shall display an error or welcome the user with their nickname (pulled from the API response).

2. **Logoff** shall invoke the Logoff Web API (previously described) and then, when the response is received, provide a display a confirmation message that the user has been logged off (or that there was an error trying to call the logoff API).

3. The **home page** (that shows all the “other” data) shall hide the insert/update/delete icons unless the user is logged in.

4. After a user logs in, they might see the home page icons but then walk away from their computer and their session might time out. So, each of these APIs have to be modified to check that the user is logged in: update other, insert other, delete other. To test this, run your SPA and log in so that you can see the icons – you might even want to spin off three browser tabs that show the Other data with all the CRUD icons. Then, you can run logoff.jsp (will emulate timing out), then test each browser tab (doing insert, update, and delete) to be sure that you are prevented from doing the operation unless you are logged in.

5. **Blog** in your Labs page that discusses your work, provides a link to tripleLab/index.html and direct links to each of the two Web APIs that you wrote this week (each link having an example URL that shows it’s working).
Submission – Check Blackboard for Due Dates

- Note: do not let problems with your logon/logoff code prevent us from being able to give credit for functions that would otherwise have worked correctly. Test carefully and be sure we can run all that works.
- Test that the source code for all of your labs is working correctly inside of a single NetBeans project - this is called regression testing. Make sure your triple lab did not clobber java classes needed by your Web API lab.
- Check that your labs page has blog for this lab, including links to your Web APIs (with hard coded URL parameters that you can use for testing and we can use for grading).
- Publish and test what you published, then submit a zip file into blackboard by the due date/time.
- Remember that publishing is more involved now that we are using java classes. Refer to the publishing instructions provided in the Web API lab if you have trouble with publishing.
- Follow the "Requirements for All Labs and Projects" at the top of the 3344 labs page.

Suggested Approach for Parts I and II:

1. Lab Activity: If you completed the first Angular Lab Activity, you should have gotten a good start with "http get and ng-repeat", "sorting and filtering" "routing for layout and with parameters". If not, study the Angular tutorial code (Sally's site, Tutorials – Front End – AngularJS).

2. Install the sample code on your development PC/MAC (in its own project) and study it. Experiment with a copy of it. Make sure to always have a working copy of the sample code installed on your PC/MAC.

3. If your project is malformed (you are getting weird compiler errors), create a new project and copy down the Source Packages and client side code (and add the two jar files to the Libraries folder). Carefully follow the instructions for installing a sample program from a zip file (you'll find these instructions in the Web API lab). If this does not help, maybe your installation is bad. If so, uninstall glassfish and netbeans, then install the latest java JDK (even if you already have a JDK installed), then reinstall the NetBeans bundle, create a new web app project, add a JSP page to the web folder, run the newly added JSP page to see if your installation is OK.

4. Set Chrome as the default browser in NetBeans (Tools – Options, Default Browser pick list) and make sure you have installed the JSON View Plugin for Chrome.

5. Write JSP Web APIs first and run them directly from NetBeans (right click Run the JSP page). If it is a Web API that expects input parameters, use URL hacking to “hard code” the input parameters. Only when the Web API is working as expected, should you begin to work on the related client side functionality. Remember that the top line of a Web API JSP page needs a special JSON content type (this is different than a JSP page that would provide a user interface).

6. Only write a few lines of code between testing. Whenever you get something working well, back up and name the backup so that you will know what it is.

7. When you are developing from home, you must be tunneled in so that your Web APIs (JSP pages) can access the DB.

8. If you see a database error in the JSON of the web page,
   - check that you have added the mysql jar file to your project's libraries (the error message would complain about mysql driver),
   - check that you are really tunneled in (the message would be something about connection),
   - check your username/password in your DbConn (message would be about invalid username/access)
   - It could show you a SQL syntax error for the delete statement, so you'd fix that in your java code.
   - You may have done a results.getObject("someColName") and tried to run that through a formatter that expects a different type. Make sure that you are formatting data according to the type that it is in the DB.
9. Do not develop on published code (YIKES!) This would be an unimaginable amount of work and very hard to debug (you wouldn’t have access to the server log to see debug messages). You must run/test your project locally and only publish when things are working.

- Use System.out.println to write debug messages while developing your Web APIs, from JSP and/or java (server side) code. You’ll see these messages in the Glassfish Server Log (in the output area below the editing area). If you are a MAC user, click on Services (instead of Projects in upper left), open "servers”, right click "glassfish server”, select "view domain server log". If your code was published, you would not be able to access the tomcat server log where your messages would be going...

10. HTML/JavaScript debugging.

- Put “debugger;” on any line of JavaScript code where you want the Chrome debugger to start, then you can single step through your code (see image of debugger on Sally’s JavaScript Tutorial).
- Use console.log to write debug messages. If you console.log an object (by itself) you’ll see an object in the console log that you can open and discover its properties and values. You can see the console output if you press F12 from chrome and click on the console tab.

11. Name your JavaScript property names exactly the same as your StringData property names (in your server side, java/JSP code) – because of "automatic conversion tools" that we use. The GSON class converts to/from java object to JSON on the server side. In the client side, JSON.stringify() and eval converts to/from javaScript object to JSON.

12. Remember that you must "Run" (not "View") the HTML pages because AJAX makes http requests which only work if there is a web server (localhost) communicating (http) with a browser. Ajax uses the request and response of the http protocol and that requires that a web server (apache) is involved serving up pages. Once you have Run a page, you can refresh to run again (after making changes). If you feel that you are not running your latest changes:

- Right click and run index.html (instead of just refreshing whatever page you were testing).
- Hold down the Control key while you refresh your browser (it is supposed to not use cached items)
- From your Browser’s menu, clear your history/cache and/or Open an Incognito Window (from Chrome’s menu icon, it’s the third option).
- Right click on your project and select "Clean and Build" (to recompile everything).

13. If you are getting a CORS error (Cross Origin Resource Sharing), you are probably using a fully qualified URL (starts with "http://" instead of a relative URL in your ajax call. A CORS error would not occur unless a page from one domain (e.g., localhost) is requesting data from another domain (e.g., your published Web API).
Suggested Approach for Week 3 (Log On / Log Off):

1. Work on your Web APIs first:
   a. The logon API can be modeled after the Web API that you made for lab 6. You can extract username/email and pw using request.getParameter(). You call a java method passing in the username/email and pw and returning a (user) StringData object. If the java method finds a user record with the given username/pw, then the JSP page needs to write something into the session (mentioned above), for example, you can write the person’s nickname (String) into the session or you can write a fully populated (user) StringData object into the session. The JSP page should also out.print information onto the page (in JSON format) – to let the page know if there was an error or not.

   
   ```java
   session.setAttribute("loggedInUser", foundUserName);
   ```
   
   b. The logoff JSP page just has to invalidate the session and then write some JSON code for the client, just to let the client know that it has done the job. It is unlikely that there would be an error with logging out since no database is needed.

   ```java
   session.invalidate();
   ```
   
   c. You might want to write a JSP page that simply says “you are logged on” (maybe with the nickname of who’s logged on) or “logged off”. This can be very helpful when debugging.

2. Log On page:
   a. Add a new Log On route path in your routing.js file. This path should specify a new logon controller (js file) and a logon partial (html file). Test that your nav bar works to show the logon partial in the content area.

   b. The Log On partial should look like any Log On page (username/email and password textboxes, Log On button, message area for messages like “welcome” or “credentials not found”. For your password input, use `<input type="password">` … instead of `<input type="text">`

   c. The Log On controller will be similar to the person insert or person update controller in that it needs to contain a Log On function that will be invoked when the user clicks the Log On button.

   d. In the Log On function (that’s in the Log On controller) needs to make an ajax call passing the username/email and password to the logon API. After receiving the AJAX response, communicate success or failure to the user (by putting a message into $scope so that the Log On partial will show that).

3. Log Off page (steps are similar to Log On):
   a. New Log Off path in routing js file that uses a Log Off controller that makes an ajax call to the logoff API. After receiving the AJAX response, the Log Off partial just needs to show a confirmation message (that the log off did happen).

   a. When the web page calls the Get Other Web API, it is going to need to know if the user is logged on or not (so it can decide if it should show the icons or not). Modify the StringDataList (POJO under Source Packages) for the “other” database table. Add a field like “loggedInUser” (e.g., next to the dbError field that is already there). The getOther Web API should check the session (as shown below) and set the StringDataList.loggedOnUser field. When you extract information from the session, remember to type cast it to the same type it was when you put it in (e.g., String or StringData). If the user is not logged in, what you extract will be null.

   ```java
   myStringDataList.loggedOnUser = (String) session.getAttribute("loggedInUser");
   ```
b. Modify the Get Other Controller (js file) so that it sets some controller/global boolean like isLoggedOn to be false. In the success function of the ajax call, set this boolean true/false based on what you have received from the ajax call.

c. In the List Other Partial, use “data-ng-show=isLoggedOn” for the <td>’s that hold the edit and delete icons for each data row. Put a span tag around the insert link and apply the “data-ng-show=isLoggedOn” attribute to that as well.

5. Modifying insert user/other, update other, and delete other. The JSP pages for these APIs must first check if the user is logged in. If they are NOT logged in, write an error message like “unauthorized” into whatever response you will write to the client AND be sure to NOT perform the requested operation.

6. TESTING.
   a. Run your index.html page. Navigate to the user list.
   b. Run the Display Other page and make sure that you are NOT seeing any CRUD icons.
   c. After making some validation mistakes (like duplicate email, not supplying required fields), register then make sure the new user shows up in the user list (check all fields that you entered).
   d. Log in as the new user you just added.
   e. Run the Display Other page and make sure that you ARE seeing the CRUD icons. Copy the URL into three more browser tabs (one for insert, one for update, one for delete).
   f. In one tab test insert other (making some validation errors first). In a second tab test update other (making some validation errors first). In a third tab, delete the row you inserted. Refresh the first tab (display other data) to be sure that all the data looks as you expect it. Then log out from this tab - this is to simulate a user timeout – they didn’t log off but they went away for some time and they are still seeing update/delete icons and the insert link.
   g. In each of the three other tabs, test insert, update, and delete. All three operations should be rejected with message like “unauthorized – you must be logged in to perform this function”. Note: you may have to disable the NetBeans connector to test like this – otherwise it might try to refresh pages for you when it thinks you need it.