CIS 3308 Update Homework

Overview:

In this homework, you will modify your web application so that your list interfaces (web_user and [other]) each provide an edit icon on every data row (see sample UI below).

![Web User List](image)

HTML/JavaScript Code:

This is how the dynamically created HTML code might look for one of the rows <tr>s of the HTML table. Use the Elements tab of Chrome F12 if you need to debug to see the HTML built by your JavaScript code. Open up the Content area and drill down until you find the HTML code you want to check.

```html
  <tbody>
    </tr>
      </td style="text-align: left;"/>
      </td style="text-align: center;"/>
    <td style="text-align: right;">$3,115.00</td>
    <td style="text-align: left;">Admin</td>
    <td style="text-align: right;">1</td>
    <td style="text-align: center;"><img src="icons/update.png" onclick="window.location.hash = '#/userUpdate/' + userList[i].userId"></td>
  </tbody>
```

Note: it is a little “tricky” to build the `<img>` tag. I used all three quotes: back tick, single quote and double quote as follow (code pulled from around line 57 of webUserList.js):

```javascript
  userList[i].update =
    '<img src="' + CRUD_icons.update + '" onclick="window.location.hash = '#/userUpdate/' + userList[i].userId+'"'>;
```

Click Sort and Filter Ability:

Although my update sample code does not demonstrate Click Sort and Filter ability on the HTML table, your code should have that (from previous homework). The update sample code just shows the new functionality - how to update.
Database Design:
- Your database homework required uniqueness constraints, null-able non character fields, and foreign keys. Make sure that your database meets all these requirements. If not, make the database design changes and test/fix any/all previous homeworks – or you will get deductions on homeworks and on the project because your code is not being properly tested against these important/representative aspects of database design.

Functionality after Clicking an Update Icon:
- When the user clicks on an edit icon (of a user or an [other] record), the UI shall display the data of the clicked row pre-filled into textboxes.
- Share as much UI code as you can between insert and update, however update will have some changes:
  - The id field (of user or [other]) shall be disabled so that the user cannot change this value (perhaps for input you may not have even shown this field).
  - The FK (of user or [other]) shall have the correct option pre-selected (based on FK value in the db).

NOTE: In the insert homework, most students will have implemented the user FK from [other] to be a pick list (similar to how role is an FK from user). However, if you elected (in the insert homework) to have the web user id set to the logged on user, then you’ll need a different approach for the update homework, as follows:
  - When building the HTML table for displaying your “other” records, only put an edit icon next to the records that were entered by the user who is logged on. This is client side “security” (which by definition is not secure, but it does prevent a well-intentioned user from trying to access records that are not theirs).
  - In the otherUpdate API, refuse to do the update unless the correct user is logged on. You can test this by having two tabs open – one that displays records when you were logged on as one user and another that displays records when you are logged on as another user. Then try to edit records for which you are not eligible and make sure you get a server side error message preventing you from doing so. This is server side security – which is actually secure.
  - Make sure that you have at least two different users “owning” [other] records and provide user passwords in the user list – to facilitate our testing (which we’ll do as described above).
Validation / Update:

Once the save button is clicked (saving user or [other]), the application shall validate each field (for conformance to field type/length rules in the database). Note: this is server side validation. In a real web application, you would probably implement Client Side (JS) AND Server Side (java) validation. Client Side validation provides quicker error messages to the user and Server Side validation would be a double check. However, since we do not have time to do both, we will just do Server Side, which is more secure and more challenging. Also, Client Side validation would “mask” Server Side validation, preventing us from being able to verify your Server Side validation.

- If the values pass field level validation, then update shall be attempted (in method model.webUser.update or model.[other].DbMods.update).
- When the update is executed, if a uniqueness constraint results in a database exception, your web application shall intercept the database exception messages and provide (instead) a “user friendly” record level error message, such as the one displayed below.
- Note: it is unlikely that the user will get a foreign key error since you provide them with a dynamic pick list that only shows valid values from the database. If a hacker URL tampers and provides an invalid foreign key value, it is OK for them to get an “unfriendly” database exception message 😞.
Server Side Design Specifications

- The Web APIs (JSP Pages) shall be as short as possible, containing only the code that **must** be in the JSP page, which is:
  - getting a database connection. (We do this in the JSP page in case the db connection is needed for several purposes – it can be reused.)
  - input (request.getParameter),
  - converting between JSON and POJOs (Plain Old Java Objects), and output (out.print).

The rest of the code shall be moved to reusable java class code.

- Your project shall have a model.xxx package where xxx is replaced by the name of your “other” database table.
  - “other” (or something equally uninformative) shall NOT be the name of your database table
  - model.other shall NOT be the name of the package that holds the DB access code for your “other” db table.

- Each sub package of the model package (e.g., model.webUser, model.xxx, etc), shall have the same named classes: StringData, StringDataList, DbMods, etc.

- DbMods (in each of the model packages) shall employ code reuse, meaning that a validation method shall be used by both insert as well as update. The validation code shall take a StringData as input (containing user input values) and provide a StringData object as output (containing field level error messages, with empty strings for fields that passed validation).

- Methods update (and insert) in class DbMods shall also take StringData (user input values) as input and provide a StringData object as output (containing Server Side error messages).

- For any nullable non character field that the user chooses not to provide, the web application shall encode null (as demonstrated by the sample code for the web_user table).
Removing Dollar Formatting before Validating and/or Converting to Decimal:

- The StringData object that’s returned from the database (that pre-fills the text boxes) formats any dollar amounts (with commas and $). If the user immediately clicks save, the user would get an error for invalid dollar amount – unless you strip out the commas and the $ before attempting to validate the users input. So, check to see if you have the code (in red box) in class dbUtils.validationUtils.

```java
public static String decimalValidationMsg(String val, boolean required) {
    if (val == null) {
        return "ValidationUtils.decimalValidationMsg(): Programmer error -
    }
    if ((val.length() == 0) && !required) {
        return ""; // Since this field is not required, empty string is v
    }
    val = val.replace("$", ""); // removes $
    val = val.replace("", "); // removes ,
    try {
        java.math.BigDecimal convertedDecimal = new java.math.BigDecimal(val
        return "";
    } catch (Exception e) {
        return "Please enter a dollar amount";
    }
} // decimalValidationMsg()
```

- You also have to strip out those same commas and $ before attempting to convert the user’s input to decimal (so that you can write it to the database). So, check to see if you have the code (in red box) in this other method, still in class dbUtils.validationUtils.

```java
public static java.math.BigDecimal decimalConversion(String val) {
    if ((val == null) || (val.length() == 0)) {
        return null; // Since this field is not required, empty string is valid user entry
    }
    val = val.replace("$", ""); // removes $
    val = val.replace("", "); // removes ,
    try {
        return new java.math.BigDecimal(val);
    } catch (Exception e) {
        System.out.println("ValidationUtils.decimalConversion(): cannot convert " + val + 
        return null;
    }
} // decimalValidationMsg()
```
NESTED AJAX CALLS:

- Another tricky issue with update is having to make two AJAX calls to build the UI to let the user see the data they want to update.
  - One AJAX call has to get the web user data and
  - another AJAX call has to get the list of roles.

Only after getting the web_user data AND the list of role ids/names, can you build the role pick list which is preselected to the role_id found in the web_user record.

To solve this problem, I nested my AJAX calls. I didn’t invoke the get role list API until after getting the web user data. Looking at the UI (in my solution), you might first see a plain text box for role, but then when the role list API comes back with the data, that is replaced with the pre-selected role pick list. There might be other ways to do this. You can experiment if you like. With web apps generally, we often notice the UI changing here and there as we look at it – these changes are probably due to different AJAX calls coming back at different times.

- NOTE: if this were a real web application, I would probably decide to get the role list once (at index.html page load time) and use that wherever needed afterwards. But, I did it this other (more difficult way) to model what you would have to do in cases where you need to get the latest data to populate a pick list.
Client Side Design Specifications

- **JS file naming convention:**
  
  *Every js file shall be named the same as the single global function (or object) that is defined in the file.*

- The client side code for insert and update [other] shall be stored in a single JS file, in a single JS object – to facilitae code sharing. To prevent pollution of the global name space, use an IIFE to wrap private functions to be shared by insert and update. Examples of functions that can and should be used by insert and update:
  - A function that bundles up the user entered data into a JSON string to become part of the Web API calls (of insert/update).
  - A function that breaks up the JSON received from the insert/update Web API and places the field level error messages onto the insert/update user interface (html partial file).
  - The User Interface for update and insert.

- The client side code for list [other] can stay in a separate file.

**Example of Client Side code for user insert/update**

Here is an outline of the user insert/update code that I have in my webUserMods.js that I shared with you as sample code for this homework assignment. When you write your insert/update code for your "other" table, you do not have to have the exact function names that I do below, but your insert and update code shall employ reuse wherever possible.

```javascript
var webUserMods = { }; // this is an empty, globally defined object. The IIFE (that runs next) will add public methods to this object.
      // Because these public methods are defined in the IIFE, they can share private functions also defined in the IIFE.
    ( function ( ) {     // This is an IIFE, an Immediately Invoked Function Expression
      // The IIFE provides a context where you can place private functions that can be shared by insert and update.

      webUserMods.insert = function ( ) { ... };     // public function, invoked by Routing when the user clicks on
          // Register link or the plus sign from User List.

      webUserMods.update = function (webUserId) { ... };    // public function, invoked by Enhanced Routing (using URL parameters)
          // when the user clicks on an Update Icon.

      // // Private functions may be called by insert or update...
      function createInsertUpdateArea (isUpdate, targetId) { ... } // creates HTML that is similar between updateUI and insertUI –
          // differences: one button calls insert save the other calls update save
          // differences: update needs disabled Web User Id row and data

      function dbDataToUI (obj) { ... }                     // private fn, after getting the data (from db), places the data object into the text boxes
      function getUserDataFromUI ( ) { ... } // bundles the user data from the text boxes into JSON to be appended to Web API call
      function writeErrorObjToUI (jsonObj) { ... } // places error object into the various error message areas on the UI
    } ());  // This closes the IIFE & immediately invokes it.
```
Enhanced Routing

Recall with insert, we had to set window.location.hash = “#/insertUser” when the user clicked on the plus sign icon from the list user page - rather than just call some function like user.insertUI( ). Here’s the reason. Suppose the user clicked on list then insert (and we did not change the link for insert) and then the user clicked on list again – nothing would happen (no list). Why? Without a change to the URL (link), the routing function does not get invoked and so the list click is ignored.

We run into a similar issue in this homework. Here is the scenario: the user clicks on list user, then they click on an update icon, then they want to list the users again. As before, we must change the URL to #/userUpdate (change the URL) so that clicking on list will invoke the list function. BUT we also need to communicate the ID of the user to be updated. For this, we need more extensive routing code, code that extracts a routing parameter (the user id) from the new URL and passes that to the update function. Note that the solution code for update utilizes routeParamFw.js (not just routeFW.js). This enhanced routing code checks each new URL to see if it has a parameter, if so, it invokes the single use component, passing in this parameter.

```javascript
// private function that will be called whenever a link is clicked (or href changed)
function router() {
    var path = location.hash;
    var ele;
    var pathObj = parsePath(path);
    if (!routes[pathObj.funcName]) { // the funcName of the URL was never registered in the routing table
        ele = document.createElement("div");
        ele.innerHTML = "<p>Error: unknown link " + pathObj.funcName + " was never added to the routing table.</p>";
    } else if (pathObj.param.length > 0) { // if this URL has a parameter after the last /
        // Invoke the function that's stored in the routing table, passing in the parameter.
        ele = routes[pathObj.funcName](pathObj.param);
    } else { // This is a “regular” URL with no parameters, so don't pass any parameters into the single use component.
        var ele = routes[pathObj.funcName](); // returns DOM element from the function stored in the routes
    }
    inject(ele);
} // router
```

pathObj has two properties:
- funcName and
- param

If there is a routing parameter in the URL, invoke the single use component function passing in that parameter.

Run the update solution code. After listing the user data, click on one of the edit icons and notice what you see in the browser’s address bar:

```
/index.html#/userUpdate/1
```

It is the id of the user to be updated. That value (1 in this case) is extracted from the URL (by the enhanced routing code) and passed to the webUserMods.update function which expects an id to be passed in:

```javascript
webUserMods.update = function (webUserId) { (8)
```
All the files in here should be the same as previous homework, except for webUserMods.js which will have new update code. You will also have a file for listing your “other” data as well as another file for “other” mods.js.

Make sure to use the new enhanced routing “FW” style code.

You should also have APIs like getOtherById, listOther, insertOther, updateOther.

Wrapper class to java.sql.PreparedStatement that does setNull when null passed into any of the set methods (setInt, setDate, etc.)
Blog

- Your blog area shall include a blog that explains what you learned (what was hard/confusing, what was easy) about this assignment.

- Please double check that your database blog links to your document that shows your database design (among other things).

Programming Style

- Adhere to the requirements listed in the section entitled "Requirements for all Labs and Project" on the class labs page.

Homework Submission

- After getting your code to work locally, publish it to cis-linux2 and test it.
- Submit a zip file of your whole project to blackboard.

Debugging Tips

- Code and then test your Web APIs first by using URL tampering. Run the Web API first without any URL and it will print out the JSON for a webUser object filled with error messages. You can then copy paste the webUser error JSON into the URL, replacing the various field error messages with data you want to try to update.
- Use System.out.print() to debug java/JSP code. Look for this output in the glassfish server log (in the output area at the bottom of the screen). Open the output area from NetBeans menu option “Window – Output”.
- Once you get your Web APIs to run as expected, begin to write and debug your JavaScript code. Always have the chrome console open (F12 or right click and inspect element). This is where you will see any runtime errors printed. Add console.log statements to help you debug. If you want to see all the properties of an object console.log - just that object (if you concatenate anything with the object, you won’t be able to open up the properties of the object).
- To check that you dynamically built the update column correctly (with clickable icon), click on the “Elements” tab instead of the Console tab from chrome F12/inspect element.

Example Deductions (out of 20 points)

- Submission:
  - -18: if no zip file submitted into Canvas.
  - -18: if code not published.

- Update Functionality:
  - -4: Did not successfully incorporate sample code for Update web user.
  - -16: Did not successfully complete the implementation of Update Other
    - -4: did not implement pick list for foreign key of “other” (should have list of user emails, should set web_user_id as value of the FK, see how sample code handled web_user FK user role) AND did not implement restriction that only a logged on user can edit “their own” “other” records. You had to implement one or the other method. (Students could choose between the two methods of handling FK.)
    - -4: update not correctly implemented for YOUR TWO nullable non-character fields (e.g., date or number).
    - -2: did not have user friendly database error message indicating duplicate value for “other” unique field.
• Server Side Design
  o  - 1: if the Update User API did not push down code to reusable java classes (i.e., your code may have been modeled after the lab activity rather than the more sophisticated sample project that was provided).
  o  - 3: if Update Other API did not push down code to reusable java classes (i.e., your code may have been modeled after the lab activity rather than the more sophisticated sample project that was provided).
  o  - 2: in model."other".DbMods, if insert and update copy/pasted validation code instead of calling a shared validation method.

• Client Side Design
  o  - 1: if client side functionality related to user insert/update is not in a single JS file, with sharable private methods inside of an IIFE (as was done in the sample code).
  o  - 3: if client side functionality related to “other” insert/update is not in a single JS file, with sharable private methods inside of an IIFE.

• Display Data:
  o  web user records:
    •  - 1: if the web user list does not demonstrate that it can handle null values (for membership fee and birthday). Make sure your database has at least one record with null for all nullable non character fields.
    •  - 1: if web user list does not show the name of the user role type.
    •  Note: this was implemented already in the sample code.
  o  other records:
    ▪  - 2: if “other” list does not demonstrate that it can null values (you were required to have two nullable non character fields like decimal/dollar, integer, date – make sure to have at least one record with null for these fields).
    ▪  - 2: if “other” list display does not show the at least one non-key field from web_user table (probably email address which identifies which user).
  o  Up to -2 if data in HTML tables is not formatted nicely and/or not aligned properly (numeric values and associated column headings should be right justified).

• Routing:
  o  -5 if routing not used (e.g., student created separate HTML pages, not injecting content into a single page).
  o  -3 if enhanced routing not used (the version that extracts a parameter from the URL, if there is one, then passes that parameter to the single use component function).

• Layout:
  o  Up to -2 if overall layout of your web site does not look professional (e.g., text too close to visible edges, text hard to read due to size or color, color scheme not subtle, layout not fluid, undesirable text wrapping at narrow browser width, large images slowing down page load). Also, your pages cannot be overly similar to sample code nor to classmate code.

• Blog:
  o  Up to -2 for lack of blog content (where you tell what was hard/easy/important about this week’s HW).
  o  -1 if your database blog still does not link to a document that shows your database design and data.

• Coding Style / Design Specs:
  o  Up to -4 for what we notice when we “View Source”:
    •  HTML/CSS/JS syntax errors which would show in red font from Firefox View Source.
    •  poor coding style. Your code should be neat, properly indented, well named, and with no unnecessary code.
  o  Up to -2 for code organization (not meeting design specs).