Web API Lab

In this lab, you will produce three deliverables in folder “07_webApi”:

1. A server side Web API (named webApi.jsp) that accepts an input parameter, queries your database, and then returns a JSON string that represents an array of objects (records) that match that input parameter. This web API will be written in java/JSP. This will be very much like the sample page: getCountryFlagsAPI.jsp

2. a (client side) html page (named testApis.html) that invokes your Web API using the AJAX technique and javaScript (no jQuery) and modifies the Document Object Model, creating a HTML table and adding rows and columns of data into that, from the JSON string. This page will be very much like the sample page: testAPI_JavaScript_DOM.html

3. a (client side) html page (named testApiJq.html) that invokes your Web API using the AJAX technique and jQuery that works functionally the same as the page testApiDomJs.html (there is no exact sample code for this – you have to do some googling to figure this out).

You will add a "Web API" blog to your labs page and link to all three of the above files. For this lab, I recommend that you work in a new web application project until things are working (see below, left), then copy in all your previous labs (see below, right).
When you have finished this lab, you will merge this project with your current project so that the source code of your web site continues to be inside of a single local project.

Note: (Almost) fully implemented sample code was provided this week, simply because there are so many new concepts (including server side programming which is not the focus of this course). Please DO NOT blindly use the sample code, modify it, and submit it as your work. If you do, chances are that your quiz and exam grades will reflect your lack of understanding. Please dig in deep and learn as much as you can from this week's lab.

Client Side Functional Requirements

testApiDomJs.html shall:
1. Provide a text box where the user can enter a search filter value and a button to initiate a search.
2. When the user clicks the button, display any possible ajax and/or database error OR display an appealing HTML table that shows all the properties of all the objects listed in the JSON (that it got back from the Web API) – all properties except each object’s errorMsg property that is for debugging.
3. Have any layout that makes sense with the API that it calls – as long as the page looks professional. The layout can be the same as a previous lab or different. If your layout for this week is different from previous weeks, create a new external style sheet just for this week.
4. Provide instructions (text on the page) that let us know what we need to do to test/grade your functionality. Pre-fill a value in the text box that will yield some results (but not too many). These instructions should not make your page look less professional because any user would need to know how to use the page.

testApiDomJq.html shall:
1. Have the same functionality as testApiDomJs.html page
2. Have the same layout and reference the same external style sheet as testApiDomJs.html.

Client Side Design Specifications

testApiDomJs.html shall:
1. When the button is clicked, use just JavaScript (no jQuery or other JavaScript library) to make an asynchronous ajax call to your Web API, passing the user input as a parameter along with the call.
2. When the ajax response comes back, display ajax and/or database error OR dynamically build a HTML table that shows the data from the ajax response.
3. Utilize the more sophisticated ajax calling code (framework style) provided in sample file ajax.js.

testApiDomJq.html shall:
1. Function exactly like testApiDomJs.html but it shall use jQuery wherever possible.
Server Side Functional Requirements

`webApi.jsp` shall accept a query parameter then display JSON that contains all the attributes of all the records of your “other database table” that match the query parameter (meaning that the record’s name/descriptive field starts with what was provided in the query parameter).

Server Side Design Specifications

`webApi.jsp` shall:

1. Extract the query parameter from the Web API call using `request.getParameter()`.
2. Open a database connection object then pass that plus the query parameter to the `StringDataList` constructor (which will return a fully populated `StringDataList` object, see below).
3. Close the database connection – this is important otherwise we’ll have database connection leaks and this can bring down the server if the load is heavy enough.
4. Convert the `StringDataList` object to JSON (using the Gson library).
5. Write the JSON (out.print) as a response to the web API call.

The Source Packages of your NetBeans project shall be organized like this:

- The `dbUtils` package shall include the classes from the sample code (DbConn, FormatUtils) with only this change: class `DbConn` shall have your database credentials instead of my database credentials (twice – once for the connection string for when you are tunneling in from home and the other for after your project is published). You can add extra functionality to FormatUtils if you need to.
- The `model` package shall have a sub-package named the same as your “other” database table. In the example above, I have a package called “model” with a sub-package named “Xxx”. When you name your package, replace “Xxx” with the name of your “other” database table.
- The `model.yourDBTableName` package shall have two classes,
  - class named `StringData` (just a bundled object that has one public String property per field in your database – plus an extra field for "field level" error messages that will help you as you develop your code).
    - All properties in the StringData class shall be strings, even if the corresponding field in your database table is not a string. This way, StringData can represent pre-validated user input (in a future lab) and/or it can represent formatted data from your database (e.g., a dollar amount with commas and dollar sign).
    - The field level error message field is a good place to put any database exceptions that may occur when parsing a row in the result set (after doing a SQL select statement), for example you might throw an exception trying to format a string as an integer.
    - If you leave all the StringData data members public, you can just use `objectName.fieldName` syntax, instead of having to use getters and setters.
- **class named StringDataList** which is an array of StringData objects plus a possible "list level" database error message (String). As an example, a list level database error could be database down, invalid sql statement – anything prior to trying to extract data from the result set. This class shall have two constructors:
  - default constructor (no input parameters passed in): an empty array and empty string for database error message.
  - Overloaded constructor with input parameters: open database connection object, query parameter. This constructor shall execute a SQL select statement that extracts all the attributes of all the rows of your “other” database table – where the name(descriptor) field starts with the characters of the query parameter.

**Program Style Requirements**

Check the "All Labs and Project Requirements" section of the 3344 labs page.

**Blog and Navigation Requirements**

As mentioned in the overview, you will add a "Web API" blog to your labs page and link to all three of the deliverable files (webAPI.jsp, testApiDomJs.html, testApiDomJq.html). If you like, you can add a link or two into your real nav bar (that is included in index.html and labs.html using Angular ng-include), but you do not have to. If your nav bar is getting too full, use dropdowns.

**Submission**

- Test locally (making sure everything still works from previous labs)
- Then publish and test what you published.
- Then submit a zip file into blackboard by the due date/time.
How to Debug Java/JSP Code

In your other java classes, you have used `System.out.println()` to print debugging messages to the Console. This is the technique that you should use in this lab as you work on getting your JSP page to print valid JSON to the browser (with help from your java classes). `System.out.println()` statements are not printed into the web page, but they are viewable in the "Glassfish Server" log tab within the Output Window (lower right of NetBeans UI). If you work on a PC, you probably see this server log already (or can easily get to it from the NetBeans menu by selecting "Window – Output"). On the MAC, you can still get to the Glassfish Server log, but you need to

- click on the Services tab (upper right, where you are used to seeing your project navigator),
- open up the "Servers" tree element, then
- right click on "Glassfish Server" and select "View Domain Server Log".

Notice that in the Glassfish Server log of the following screen capture, you are seeing messages like "JSP Page ready to search for country with A" – this was generated by `System.out.println()` statement in the JSP page. This is followed by "Searching for countries that start with A" which was generated from one of my java classes.
**Suggested Approach**

1. If you have done the Web API lab activity (that is described on the 3344 labs page, just prior to this Web API lab), you should have a sample NetBeans project that can access my database (but you must tunnel in if you are working on a non-Temple PC/MAC). If you have not done the Web API lab install the sample code (provided from this lab on the labs page), tunnel in if necessary, and run the JSP page locally (getCountryFlagsAPI.jsp).

2. Leave my sample project “as is” so that my project always works, so you can use it for reference, as you debug your code.

3. Create a second project, and name it with your last name in it (this will become your new primary web app). Copy in my sample code (add the two jar files to the libraries folder, copy/paste the packages, copy/paste the HTML/JSP and any auxiliary files to the “Web Pages” Folder.) Rename getCountryFlagsAPI.jsp to be **webApi.jsp**.

4. In your project, add a new package named “model.yourDbTableName” (substituting the name of your database table).
   a. Copy over the StringData and StringDataList classes from model.countryFlags into your new package.
   b. Change the field names in the new StringData (so that they match the field names of your database table) – remember all data members should be of type String even if the corresponding data type in the database is not string. This is because StringData will be used to hold pre-validated user input data and/or formatted output data.
   c. Change the code in StringDataList so that it reflects the SQL select statement that you need to get records from your table. Fix compiler errors (in StringDataList) that were created by your changing the names of the data members of StringData.
   d. Change the connection string in dbUtils.DbConn to be your credentials. You have to change it in two places because there are two connection strings – one for when you are running from home (tunnelled in) and one for when it runs on the web server after publishing (no tunneling used).
   e. Edit the webApi.jsp page so that it imports from your model.whatever package (not my model.countryFlag package). At this point, I think you should be able to run webApi.jsp, but only if you are tunneled in (if you are using a non-temple PC/MAC). When you run webAPI.jsp, if your JSON comes out ugly have not installed the JSON View Chrome plugin, do that so you can see nicely formatted JSON in your browser, instead of it being all jumbled together. JSON View actually provides JSON validation as well as formatting.
   f. Keep working until your JSON data shows no dbError at the top. If you have not already stumbled across a list level error in the dbError property (e.g., database unavailable, SQL syntax error), generate one on purpose just so that you understand a bit better how the code works.
   g. Check test for field level errors (would show as the last property of the record objects that are array elements). If you have not already stumbled across a field level error, you can force a “field level error” by trying to extract a column that does not exist in the result set (in class StringDataList) or by trying to extract data of the wrong type.
   h. Make sure that everything in the server side code is named well. Where I have named things generically (like recordList), you can leave those names alone, but do not leave any reference to names like CountryFlag (or anything non-generic) in your code.
   i. Test your web API by “URL Hacking” (type something like “webApi.jsp?q=A” at the end of the URL in your browser).
5. Next, begin to work on the client side.
   a. Try to get **testAPI_javaScript.html** to work. Note: this is not required for submission but it is a good exercise.

   - To make things easy on yourself, be sure that your javaScript property names (for your database fields) match exactly the java property names in your StringData object. This is because gson goes by the field names – if they match, you do not have to code up all the name mappings.
   - Make sure that you are Running your code (from NetBeans) not Viewing it – you need to see a URL that starts with “localhost” in the browser (or else the HTTP requests will not work).
   - Use “console.log()” profusely as you are debug your javaScript. You can also do something like this: `console.log(object)` – in the console area, you will see the object and you will be able to open/close it like a tree view. Make sure to always have the chrome debugger open (F12 from chrome).

   b. Rename testAPI_javaScript_DOM.html to be **testApiDomJs.html**. Note that this page is pretty similar to your goal for this week’s lab. Use the same techniques (names, etc) as you used for the above page.

   c. Rename testAPI_jQuery.html to be named **testApiDomJq.html**. Note that you will have to do some internet research to learn how to create DOM elements like table, tr, td using jquery instead of javascript.

   - Using jQuery can be a little daunting, because an error can occur in “jQuery land” instead of your code. However, you should see a stack trace and be able to determine where (in your code) the problem actually started. Remember that you can do a console.log() even within your jQuery code—this is because you are actually writing javaScript which is just calling the jQuery javascript library.

6. After backing up your web application (e.g., that you had from previous labs) and after backing up your project that you just got to work, merge the two projects together (I guess you could copy either way). Make the layout of your two new html pages (testApi...html) have the same look and feel as your index and labs page.

7. Add a blog to your labs page and link to the two new testApi...html files, as well as to the webApi.jsp page (should just show raw JSON).

8. Publish. PLEASE PLEASE follow the publishing instructions very carefully. If you are sloppy with where you place your files, you can get yourself really stuck and I’ll have to help you get unstuck. These are some common errors:

   a. Student publish their “.java” (source) files instead of the “.class” (compiled) files.

   b. Students forget to do the copy/delete/wait/paste the web.xml file after they have copied up their “.class” files. When you publish, you copy up web.xml, not sunweb.xml. There’s nothing much in the web.xml file, but it contains settings for the tomcat JSP application server and it lets tomcat know that it should go out and check for newly published classes. Otherwise, tomcat (being geared towards production, not development) assumes that no changes have occurred to class files (it is uses a copy of what you published previously).

   c. If you ever find that you are unable to delete any files under your web root folder, email me right away, I forward your email to our CIS department IT administrator and he resets your privileges. For this, you could even email him directly: stauffer@temple.edu. Be sure to explain your problem well. He’ll need to know your NetAccess user name and the folders/files you cannot delete on cis-linux2.