CS144: Web Applications

Project Part 4: *Building a Website*
Due on Friday, ~~February 27~~ March 1, 2015, by 11:00PM

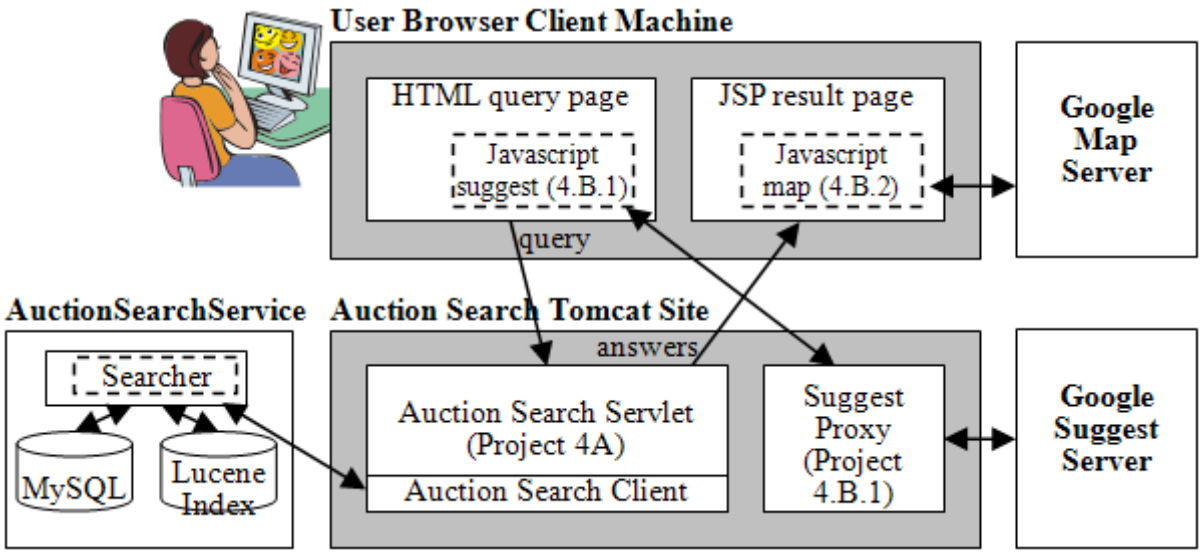
- **Submission deadline:** Programming work is submitted electronically and must be submitted by 11:00 PM. However, we recognize that there might be last minute difficulty during submission process, so as long as you started your sumission process before 11:00PM, you have until 11:55PM to completely upload your submission. After 11:55PM, you will have to use grace period as follows.
- **Late Policy:** Programming work submitted after the deadline but less than 24 hours late (i.e., by next day 11:00 PM) will be accepted but penalized 25%, and programming work submitted more than 24 hours but less than 48 hours late (i.e., by two days after, 11:00 PM) will be penalized 50%. No programming work will be accepted more than 48 hours late. Since emergencies do arise, each student is allowed a *total* of four unpenalized late days (four periods up to 24 hours each) for programming work, but no single assignment may be more than two days late.
- **Honor Code reminder:** For more detailed discussion of the Honor Code as it pertains to CS144, please see the Assigned Work page under *Honor Code*. In summary: You must indicate on all of your submitted work *any assistance* (human or otherwise) that you received. Any assistance received that is not given proper citation will be considered a violation of the Honor Code. In any event, you are responsible for understanding and being able to explain on your own all material that you submit.
- **Reminder:** Projects must be completed individually or by a team of two.

Change History

Overview

In Project 3, you developed the AuctionSearch Web service to provide remote access to the eBay data. While a SOAP-based Web service is a great way to allow other *applications* to access your data, it is not necessarily suitable for an *end user*; the user has to write a SOAP client program to access your data(!) In this part of the project, you will make the eBay data easily accessible by end users through a Web browser, by implementing a Web site on top of the Web service.

The Web site that you will implement will integrate three Web services: the SOAP-based Web service to the eBay data (built in Project 3), Google Maps (to display the location of an item) and Google suggest (to help users formulate keyword queries). The overall architecture for this project is shown in the diagram below:



Note that the bottom-left rectangle is the AuctionSearchService that you implemented in Project 3. We will make the same AuctionSearchService available at *http://oak.cs.ucla.edu/axis2/services/AuctionSearchService*, so that you can continue with this part of the project without relying on your implementation of Project 3. **Please make sure to use the oak service for your final submission and demonstration.** Your own Web service will **NOT** be running locally on our grading system. If your code does not use oak, it will not work on our grading system and

you will get significant penalty. The two gray rectangles in the middle are what will be implemented in Project 4, which integrates the AuctionSearch Web service, Google Map and Google Suggest servers.

By completing this project you will be able to learn:

1. How to build a Web site using Java servlet and JSP
2. How to access remote Web services in SOAP and AJAX
3. How to build dynamic user interfaces in Javascript

We will be providing relevant tutorials and references to help you learn the above topics.

Part A: Implementing Your Basic Web Site

In Part A, you will be making your eBay data accessible to Web browsers through the following two URLs:

(1) `http://localhost:1448/eBay/search?q=xxx&numResultsToSkip=yyy&numResultsToReturn=zzz`

(2) `http://localhost:1448/eBay/item?id=xxx`

For the first URL, your application should return the list of ItemID and name pairs that contain the keyword(s) "xxx" in their name, category or description attributes. "yyy" indicates the number of items to skip from the beginning of the list and "zzz" is the maximum number of items to return (if 0, it should return all items). For the second URL, your application should return the detailed information of the item whose ID is "xxx". For both interfaces, you should format the results in HTML, so that the results are easily viewable by a Web browser.

A.1. Learning Application Development on Tomcat

In Project 3, you used Tomcat (together with Axis2) to publish an existing Java object as a Web serivce. In this part of the project, you will use the Tomcat server in a more popular context, i.e., serving static and dynamic Web pages for Web browsers. As you may know, most web applications/sites consist of a fair number of files (like .html, .xml, .jsp, .jpg, .class, ...). In order to simplify the deployment of a Web application, Tomcat provides a mechanism to put together all files needed for an application in a single file, called *Web Application Archive*.

First, learn how you can develop and create your own Web application archive file for your application by reading the following tutorial:

- *Developing a Web Application on Tomcat*.

Before you proceed to the next step, make sure you understand how to develop and deploy a Web site using Java servlet, JSP, and Web application archive from the above tutorial.

The *project4.zip* file contains the basic skeleton code for this project. It contains the `AuctionSearchClient` code (that we provided in Project 3) to help you access AuctionSearchService. It also contains three servlet classes, `SearchServlet`, `ItemServlet`, and `ProxyServlet`, that you will have to implement as part of this project. Go over the included `build.xml` file to understand what it does. Pay particular attention to the target "build", which essentially does the WAR file packaging for your eBay application. Note that all the files in `WebContents` directory are automatically added to the WAR file, including `WebContents/WEB-INF/web.xml`. Go over the `web.xml` file to see how request URLs are mapped to the provided servlet classes.

A.2. Implementing the Search Interface

Now that you understand the structure of a Web application archive file and how Java servlet and JSP pages can be used to serve dynamic contents, implement the Java servlet that provides the search functionality for the eBay data.

1. First, create an HTML page, `keywordSearch.html`, in your web application which contains a form where the user can enter and submit a keyword query. The form's action should point to your Java servlet available at *http://localhost:1448/eBay/search* and pass three parameters: 1) `q`, whose value contains the set of keywords the user types, 2) `numResultsToSkip`, whose value specifies how many items to skip from the beginning, 3) `numResultsToReturn`, whose value tells how many items to return (or display). Let users enter only the value for `q` and let the form automatically generate appropriate values for `numResultsToSkip` and `numResultsToReturn` for nice display. Note that the provided `web.xml` file maps the request to *http://localhost:1448/eBay/search* to the `SearchServlet` class.

   Important notes: *For all links in your web application, including form actions, you should use relative URLs instead of absolute*. The reason is simple: if your links are something of the form "http://localhost:1448/MyWebApp/MyServlet/", they will *all break* if you make even a minor, unrelated change like switching Tomcat's port from 1448 to 80.

2. Implement `SearchServlet` to extract the value of the parameters `q`, `numResultsToSkip`, and `numResultsToReturn` from the request and retrieve the matching results from the eBay data.

Remember that for your final submission, you will have to modify the ENDPOINT_URL in `AuctionSearchClient.java` to *http://oak.cs.ucla.edu/axis2/services/AuctionSearchService* to use the oak service for this project. Once the URL is changed, your servlet code will access oak Web-service functions simply by calling the methods of the `AuctionSearchClient` class.

Notes on CLASSPATH: (1) When you implement a Java servlet, your code will depend on the Tomcat servlet library, `$CATALINA_HOME/lib/servlet-api.jar`. To avoid the "class not found" error during compilation, you need to pass the jar file location as the classpath to your Java compiler in your Ant script. However, you should not include `servlet-api.jar` or any of the other standard Tomcat jars in your .war file's `WEB-INF/lib` folder. These libraries are already available to all servlets running within Tomcat (all jars in `$CATALINA_HOME/lib` are), and may actually cause classloader problems if included in your war file.

(2) The `AuctionSearchClient` class in depends on the Axis2 library files. To avoid any "class not found" error during compilation, the Axis2 jar libraries at `$AXIS2_HOME/lib` should be passed to your Java compiler as a classpath. The library files are already available within Tomcat, so you do not need to include them in your war file.The above classpath issues are automatically taken care of if you use our provided `build.xml` and use the target "build" to compile and package your application.

3. Forward the results of the query to a JSP page, which will display them to the user in a reasonable fashion.

The provided `build.xml` has another target "deploy" that copies the .war file built from the target "build" to `$CATALINA_BASE/webapps`. You can use this target to copy your packaged WAR file to Tomcat for testing. Take a look at the *Ant tutorial* if you do not understand the provided Ant script well.

A.3. Implementing the Item Interface

Similarly to the search interface, now create a static HTML page, `getItem.html`, and implement `ItemServlet` class that will return a page containing all of the details of an item (ItemId, name, start and ending time, bid history, etc.) given its ID. The servlet is configured to be available at *http://localhost:1448/eBay/item* in the provided `web.xml` file. Your implementation should take the input parameter `id` whose value is the itemID of the item that you should return. When implementing this servlet, make use of the SOAP service `getXMLDataForItemId`, which returns the reconstructed XML for a particular item. Take this XML and generate a web page which displays this information to the user in a reasonable manner; for example, bids are sorted in chronological sequence: the latest bid comes first. One way of doing this may be to parse the XML into a JavaBean class of your own design, then pass that object to a JSP page for display. Do not simply show the user an XML dump. Also, do not generate the HTML output from your servlet directly. Instead, forward your request from the servlet to a JSP page. In a nutshell, you should assume that web pages are designed for web users so that you must conceal non-related messages; for instance, debugging information.

A.4. Adding Navigational Links

The two interfaces that you just implemented allow the user to discover items that are relevant to a given keyword and to access the detailed information of an item. However, the site is still difficult to use because (1) the two interfaces are completely disconnected and (2) the user always has to "go back" to the HTML pages to issue a new query.

Improve the usability of your site by adding the following navigational help.

1. On the keyword-search result page (generated from your Search Interface), create clickable links for the ItemIds of each returned item. These links should result in the user receiving a page containing all of the details of that item.
2. On the keyword-search result page, create **Next** and **Previous** links to allow users to navigate through next pages or previous pages of search results.
3. At the top of the keyword-search result page, include a text input box and a submit button by which the user can issue a new keyword-search query.
4. At the top of the item page (generated by your Item Interface), include a text input box and a submit button by which the user can input a new itemID to get its information.

Hints on link-based browsing:: Note that your servlet can be called directly through a URL in the following way:
`http://localhost:1448/eBay/myServlet?param1=value1&param2=value2&param3=value3`
where the string after the question mark contains parameters passed to your Java servlet in the `GET` method. So you can implement link-based navigation by generating result pages containing hyperlinks of the above format, with appropriate parameters that instruct your servlet to return the right result.

Part B: Adding Maps and Query Suggestion

Now that you have implemented the basic functionality of your Web site, it is time to add more help to the users by adding more dynamic UIs. Before you proceed, read about *JavaScript* as that is the language used throughout

Part B. You may also find the Javascript debugger *Firebug* (for FireFox) helpful in debugging your javascript code for this project.

B.1. Adding Google Maps

On your item page, add Google Map to visually display the location of the item. Sometimes, To use Google Map on your site, you may need to signup for a *Google Maps API key*. **Where they ask for your web site URL, you'll need to enter `http://localhost:1448`.** Once you are assigned a key, you may use it in your web pages. However, for the simple(basic) Google Map API, you do not need an API key to use it, for instance, a sample HTML page like the following:

```
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="initial-scale=1.0, user-scalable=no" />
<style type="text/css">
  html { height: 100% }
  body { height: 100%; margin: 0px; padding: 0px }
  #map_canvas { height: 100% }
</style>
<script type="text/javascript"
    src="http://maps.google.com/maps/api/js?sensor=false">
</script>
<script type="text/javascript">
  function initialize() {
    var latlng = new google.maps.LatLng(34.063509,-118.44541);
    var myOptions = {
      zoom: 14, // default is 8
      center: latlng,
      mapTypeId: google.maps.MapTypeId.ROADMAP
    };
    var map = new google.maps.Map(document.getElementById("map_canvas"),
        myOptions);
  }

</script>
</head>
<body onload="initialize()">
  <div id="map_canvas" style="width:100%; height:100%"></div>
</body>
</html>
```

To see what will happen in the above example: *click here*. You should copy the relevant parts of this sample into the JSP page you use for processing results. This example uses latitude and longitude to specify where to center the map. You'll need to browse the *Google Maps API* to discover how you can translate a location string like "Los Angeles, CA" into a set of coordinates to display on the map. This type of address to coordinate mapping is typically referred to as "geocoding".

For some "locations" you may not find a valid set of coordinates (e.g. locations like "PAYPAL OK"). For these, you may choose to simply not display a map, display a generic map such as of the entire world, or something else. Whatever you choose, it should be user-friendly, so avoid showing cryptic error messages or alert windows.

The basic map you just create must include the required limited functionality, in paticular, navigation and zoom-in/zoom-out, but feel free to spice things up by adding animations, information overlays, additional map controls, and more.

B.2. Adding Query Suggestion

Your auction site users may not know exactly what they want to search for, so in this last part you will be helping them formulate queries using *Google Suggest*. Try out their query interface to get a feel for what it does. When you are finished with Part B.2, the keyword-query box on your Auction Search web site should behave similarly to the one on Google suggest.

The Google suggest interface uses AJAX to "silently" submit requests to a *REST-style Web Service* in response to user input (keypresses). The replies are parsed, and the returned suggestions are used to populate the drop-down list of suggestions. As we learned in the class, this interaction takes place using the XMLHttpRequest JavaScript object. Before you go any further, be sure you've read the required reading about *AJAX*, *JavaScript*, *DOM*, and *XMLHttpRequest*.

B.2.1. Developing a Google-Suggest Proxy

Due to the asynchronous nature of AJAX, security is naturally a big concern when using JavaScript. To reduce some security risks, *JavaScript security* enforces a "same-origin policy", which means that the scripts on a page may only access information within the same domain. For example, a script on a page like "http://localhost:1448/MySite/index.html" cannot make a request for information from, say, google.com.

As a result, you will need to write a proxy (which runs on your Tomcat server) to pass requests between your query interface page and Google's suggest service. This proxy can be implemented by the `ProxyServlet` class that we provided which is configured to accept HTTP GET requests at *http://localhost:1448/eBay/suggest*. Your implementation should extract the passed-in query string, issues a request to the Google suggest service for that query (at `http://google.com/complete/search?output=toolbar&q=<your query>`), and returns the results back to the original caller. You may find the Java class *java.net.HttpURLConnection* helpful in building your proxy. Note that your servlet should return the *exact* XML data received from Google. You should thoroughly test your proxy (by issuing REST-style requests to it and checking the results) before moving on to the next step. Also note that, as your servlet should be returning XML, your browser may not render it properly (you may see just a blank page). Check the 'view source' option in your browser to see the data. Finally, if you intend to use responseXML to access Google suggest responses as an XML DOM, make sure that your proxy sets the "Content-Type:" HTTP header field to "text/xml" in its response. Otherwise, your browser wouldn't be able to recognize the response as an XML data, and won't be able to set responseXML correctly. You can set the content type of a Java servlet reponse by calling the `setContentType()` function of the response object.

B.2.2. Developing a Google-Suggest Client

After you have a functioning proxy, you'll need to write the "client" code in your query interface page which issues requests to your proxy as the user types and shows a list of suggestions. To do this, you will need to write a set of *event driven* (asynchronous) functions in JavaScript which submit requests and process results by populating a drop-down text box with a list of suggestions. By asynchronous, we mean that these functions are invoked as the result of "events", such as user key presses and data arrival, rather than by some control flow as is typical in procedural programming.

While there exist a number of excellent javascript libraries (e.g. *jQuery*, *Dojo*, and *Yahoo! UI*) that provide javascript implementations of common AJAX user interfaces like tooltips, auto-completes and drop-down-boxes, you need to provide your own implementation of drop-down box for this part of the project. (it is okay to use existing javascript libraries for other parts of the project if you want to.) For the creation of a dynamic drop-down box in javascript, the following pages provide an excellent tutorial:

- ~~*Creating an Autosuggest Textbox with JavaScript, Part 1*~~ (*Local mirror*)
- ~~*Creating an Autosuggest Textbox with JavaScript, Part 2*~~ (*Local mirror*)

Go over the above tutorial to learn about the HTML elements and their associated events to create a drop-down-box like interface for query suggestion. The tutorial provides a number of code snippets, which you are welcome to use for your implementation. Your implementation of Google Suggest should behave similarly to the one on Google Suggest. **In particular, as the minimum requirement, you must ensure that as the user types input, a list of suggestions are populated based on the partial query, which the user can navigate with the mouse pointer or up/down arrows and choose with the enter key or mouse click button. Moreover, under no circumstance do your code crash when a user is navigating the list.**

Remember that, when you issue a request to your servlet, you'll likely be encoding the user's (partial) query in the URL. You need to "escape" special characters in URL. For example, a space in a URL should be converted to "%20". You do not need to manually do this replacement - most "web" languages, like javascript, contain functions to do the proper URL encoding for a string, which you simply need to call.

Additional Notes for Project 4

For all of Project 4, the mentioned functionality is the minimum requirement. You are free, and of course encouraged, to add additional features, make the page more user friendly, etc. Also, you are not required to implement things exactly as indicated above. You may choose to, for example, display item details in a popup bubble using some AJAX UI library instead of on a separate page.

You must, however, abide by the requirement that your displayed result pages must be for "human consumption" - not just XML data. Also, your site should handle errors "gracefully". Your site should not show a cryptic Java error message to the user. Finally, your should not use an existing javascript library for Part B.2 of this project.

Showing your Website

By now you should already have a working website, and in this section you are going to prove it. In order to do this, you are asked to use the ***Selenium*** project, specifically, the *Selenium IDE*.

What is Selenium IDE?

Selenium IDE is designed for automating browser tests. It is essentially a Firefox browser plug-in. Primarily, it is for automating web application tests. However, here we use Selenium as a "screen action recorder". That is, Selenium will allow you to record a sequence of actions on your website, and the grader will be able to **replay** such actions flow.

Installing Selenium IDE

1. Install ***Firefox*** if you do not have in your host machine. Note that you should **NOT** install it on your virtual machine. The VM will function as the web server, and your host machine will be the client trying your website. (Please download the latest released version).
2. Download the latest version of the ***Selenium IDE***. In your Firefox browser, when you click the download link, it will pop up a window to proceed with the installation.
3. Check out the following video tutorial:

Using Selenium IDE

1. `t1.html`: use your search page to search for **superman**. Then you should click the "next page" and "previous page" in your search result. Type "superman" again in the search box in the search result page. The next step is clicking the first item in the "superman" searching result (it should be *a link pointing to a page that displays information about the item 1049430907*).
2. `t2.html`: use your search page to search for **superman** and click the second result(1045823269)
3. `t3.html`: use your item search page to search for **1045348061**. In the result page, search for **123456789** .
4. `t4.html`: use your item search page to search for an empty query(click the search button directly)

What to Submit

In this project you are creating a web site, consisting of both static content (HTML pages, images, etc.) and dynamic content (JSP pages, Java Servlets). This content is wrapped up as a Web application archive (.war) file which can be deployed on Tomcat.

Your submission should consist of a single **project4.zip** file:

```
project4.zip
 |
 + README.txt
 + build.xml
 + TestCases
 |   + t1.html
 |   + t2.html
 |   + t3.html
 |   + t4.html
 + WebContents
 |   + keywordSearch.html
 |   + getItem.html
 |   + additional files (*.html, *.jsp, *.css, *.js, image files)
 |   + WEB-INF
 |       + web.xml
 + src
 |   + java source codes for the servlet (with your own naming/structure)
 |
 + lib
     + external java libraries used (not available in our VM)
```

The `build.xml` in the zip file should have the target "build" that builds your web site into a single .war file, and the target "deploy" that deploys that .war file on the Tomcat server pointed to by the environment variable `$CATALINA_HOME`. That is, **we should be able to simply unzip your submission and run "ant build" and "ant deploy" to deploy your web site on our machine.**

Note that your final submission should include *all* files for this project. Add any additional notes or comments that you think will be helpful to the README.txt file. Once your submission file **project4.zip** is ready, submit via our submission page at ***CCLE*** .

As always, remember to allow sufficient time to prepare your submission once your work is complete.

Testing of Your Submission

The "grading script" for your submission is the file *p4_test*, which can be executed like:

```
cs144@cs144:~$ ./p4_test project4.zip
```

Add the path to the zip file if necessary after downloading the script and set its permission appropriately.

You **MUST** test your submission using the script before your final submission to minimize the chance of an unexpected error during grading. Please **make sure that your submission uses oak service**, not your own local Web service. When evertything runs properly, you will see an output similar to the following from the grading script:

```
Stopping tomcat server if it is running...
Running 'ant build' to build your war file...

   ... output from ant ...

Removing existing eBay application files on Tomcat...
Deploying your eBay application...

   ... output from ant ...

Now your Tomcat server is running with your application.
Please access your application through your browser.
Make sure that all application functionalities are working fine.
Don't forget to stop Tomcat server once you are done.
```

After you run the script, make sure to check the functionality of your site using the Firefox browser in the VM and stop the Tomcat server when you are done.

Grading Criteria

Overall grading breakdown is as below and each feature will be tested with simple cases and tricky cases.

- Servlet/JSP (40%)
- Google Map (20%)
- Google Suggestion / Search Interface (40%)