Connecting to Websites Programmatically with Android

Brent Ward

Hello! My name is Brent Ward, and I am one of the three developers of HU Pal. HU Pal is an application we developed for Android phones which attempts to make the most commonly used features of Harding's Pipeline website easily available for students.

Although Pipeline is a very useful website for students as-is, one of its biggest flaws (in our opinion) is its navigation – for a student to check their chapel information, they must click on the "Home" tab, then "Student Services," then "Student Records," then "Chapel Info," then select the current semester from a dropdown menu, and finally click a submit button! That's six or seven sequential clicks within sub-menus just to navigate to one page, and it's easy to see how this process would be confusing to incoming freshmen. We wanted to simplify this process into a single button press, and now, we have!

The following is an account of how we were able to programmatically connect to Harding Pipeline and manipulate the data to suite our needs. Although I will be showing example code that specifies how to connect to Pipeline, the methods and ideas that we used are robust enough to be tailored to just about any website you may wish to connect to. I hope this will serve as a useful tutorial for those wishing to make Android applications that access the web!

Step One – Check for Internet Connection

The first step is simple enough – before we attempt to connect to the internet, we need to ensure that the user actually has internet access in the first place. For this, we will create a simple Boolean function as follows:

```java
private boolean haveNetworkConnection() {
	boolean haveConnectedWifi = false;
	boolean haveConnectedMobile = false;

	ConnectivityManager cm = (ConnectivityManager)
	getActivity().getSystemService(Context.CONNECTIVITY_SERVICE);
	NetworkInfo[] netInfo = cm.getAllNetworkInfo();
	for (NetworkInfo ni : netInfo) {
		if (ni.getTypeName().equalsIgnoreCase("WIFI"))
			haveConnectedWifi = ni.isConnected();
		if (ni.getTypeName().equalsIgnoreCase("MOBILE"))
			haveConnectedMobile = ni.isConnected();
	}
	return haveConnectedWifi || haveConnectedMobile;
}
```

As you can see, this function initially creates two Booleans (haveConnectedWifi and haveConnectedMobile) which are set to false. We establish a ConnectivityManager and test to see if the phone is connected to Wi-Fi or a mobile data plan provided by a service carrier. If the phone has either kind of internet connectivity enabled, the associated Booleans are set to true.

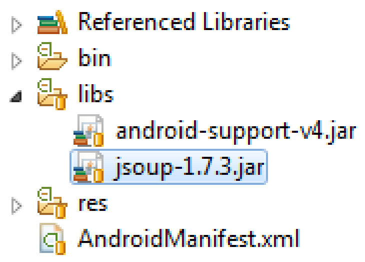
It’s a good idea to use this function any time you are about to try to connect to the internet. The pseudo-code would look something like:

```
if (haveNetworkConnection())
      try to connect to the internet;
else
      inform the user that internet service is temporarily unavailable;
```

Step Two – Download and Install the Jsoup Library

Now that we have a way to test for internet connection, we should gather some tools to make the process of logging in to the target website easier for us. For this, my team used the Jsoup library. Jsoup is an open source java library filled with useful methods and classes for HTML parsing. Unlike webviews, which simply display a web browser within an app, Jsoup is used to extract specific data from the HTML within a web page. If you wish to check a webpage to see if a specific username and password are valid, for instance, Jsoup is the perfect tool to use.

Jsoup can be downloaded for free at http://jsoup.org/. To allow your application to use Jsoup, simply add the Jsoup jar file to your project’s library folder, like so:



This will allow you to use Jsoup methods and classes in any file within your application’s project. If adding a Jsoup method still results in an error, make sure to hit ctrl+shift+o to automatically add Jsoup to the file’s imports.

Step 3 – Attempt to Login

Once you have Jsoup up and running, we can attempt to login to the website in question. The code I provide here is specific to Pipeline, but you may modify the code to work on other websites without much hassle.

First, you must obtain a username and password from the user. This can be done easily by taking the text from two edittextboxes as strings when the user clicks a submit button. Once you have the username and password as strings, it is important to launch the internet connection in an asynchronous function. Internet connections are not instant, and if we try to connect without using an asynchronous thread, we will lock up the UI thread rendering all other buttons and objects unresponsive until the connection goes through.

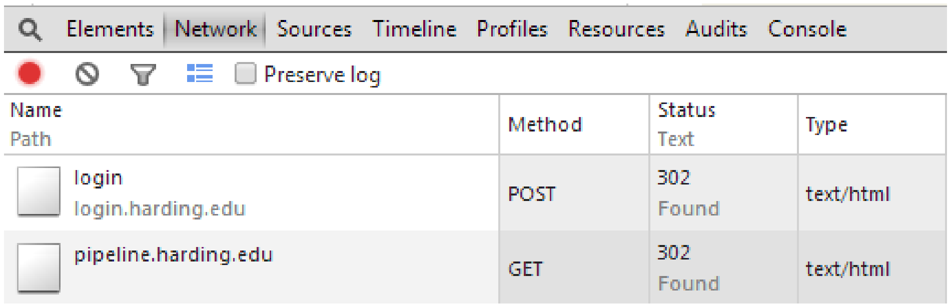
```java
private class FetchItemsTask extends AsyncTask<Void,Void,String> {
```

Within this asynchronous task, we will define a pre-execute method which will run automatically before we attempt to connect to the internet. It will display a progress dialog box which shows a spinning wheel and the text "Logging in…" This shows the user that progress is being made on a function that will take a brief moment to finish.

```java
@Override
protected void onPreExecute() {
	// Create a Progress Dialog (Spinning Wheel dialog) when user is already
	// logged in, and is being redirected to the Home page
	pd = new ProgressDialog(getActivity());
	pd.setMessage(getString(R.string.logging_in));
	pd.show();
}
```

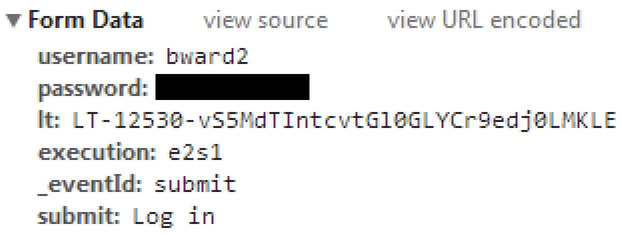
Now that the user knows we are trying to login to Pipeline, we will attempt to login using the username and password they have provided. But first, we need to know what Pipeline actually looks for when a user attempts to login.

To do this, my team logged in to Pipeline and used the JavaScript Console to view the login post method. If you want to try this yourself, login to Pipeline using Google Chrome as your browser and hit ctrl+shift+J to open the console. Once the console is open, click on the Network tab towards the top-left of the console. You should find a file at the very top of the list entitled "login" (if you don't see this file, logout and log back in with the console open to try again). It should look like this:



Name Path	Method	Status Text	Type
login login.harding.edu	POST	302 Found	text/html
pipeline.harding.edu	GET	302 Found	text/html

Once here, click on the login file and you should see the HTML that records the user's login attempt. What we are looking for here is the Form Data. Scroll down until you see this:

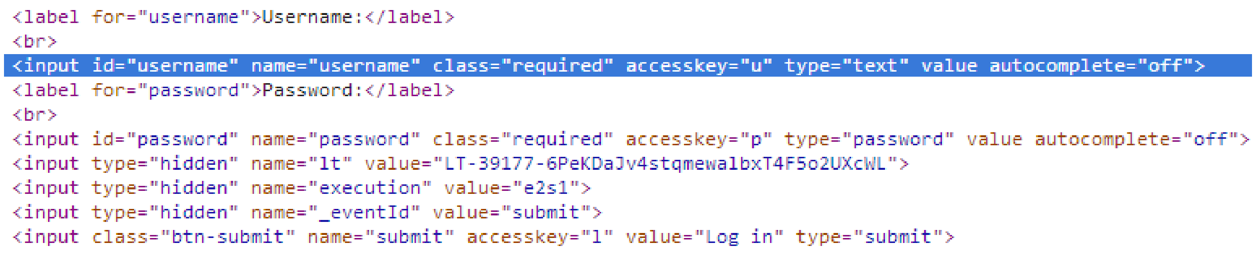


```
username: bward2
password: [redacted]
lt: LT-12530-vS5MdTIntcvtG10GLYCr9edj0LMKLE
execution: e2s1
_eventId: submit
submit: Log in
```

This shows everything that we need to ensure a successful login! Here, I have taken a photo of every value that gets submitted when I login the Pipeline. This includes my username and the password (which I have edited out of the photo – usually, the password is shown in the clear!). There is also an _eventID which shows I have attempted to submit something, and a submit field which shows I attempted to login.

However, there are two mysterious fields present here: the lt value and the execution value. If you look at the photo above, they appear to be completely random strings of characters. However, if you return to Pipeline's login page and examine the console, you may observe some sneaky hidden fields that Pipeline automatically generates.

After opening the console, click on the magnifying glass in the top left corner. This will allow you to inspect the HTML behind elements on the page. After clicking on the magnifying glass, click on the textbox where you would type your username and the console will automatically show you this:



```
<label for="username">Username:</label>
<br>
<input id="username" name="username" class="required" accesskey="u" type="text" value autocomplete="off">
<label for="password">Password:</label>
<br>
<input id="password" name="password" class="required" accesskey="p" type="password" value autocomplete="off">
<input type="hidden" name="lt" value="LT-39177-6PeKDaJv4stqmewalbxT4F5o2UXcWL">
<input type="hidden" name="execution" value="e2s1">
<input type="hidden" name="_eventId" value="submit">
<input class="btn-submit" name="submit" accesskey="l" value="Log in" type="submit">
```

Here, we can see the name of the input field for the username is "username" and the name of the password input field is "password". Simple enough! But underneath these, we see a few hidden input values. Namely, we can see those sneaky lt and execution values from before. If you refresh the page with the console open, you will see that these values are different each time the page loads. I am not an administrator of Pipeline, but I'm assuming that these random fields serve as an extra layer of security somehow. So to login, not only do we have to take a username and password, we have to allow for randomly generated strings that refresh at each page as well!

Fortunately, this is actually pretty easy to implement. The hard part was figuring this out in the first place! The next method within our asynchronous task will start as so:

```
@Override
protected String doInBackground(Void... params) {
	try {
		Response loginForm = Jsoup.connect(mLoginURL)
			.method(Method.GET)
			.timeout(10000)
			.execute();
```

This will attempt to establish a connection to Pipeline's login page, where mLoginURL is a string variable for the URL of Pipeline's login page. For the sake of your code, you may change the value of this string to the URL of whichever webpage you need to connect to. The timeout(10000) portion of code allows 10,000 milliseconds (10 seconds) for the connection attempt to go through before it times out.
To proceed with the code, we do this:

```
Document doc = loginForm.parse();
String ltValue = doc.select("input[name=lt]").attr("value");
String executionValue = doc.select("input[name=execution]").attr("value");
```

This saves the HTML of the login page as a document which we may manipulate at our leisure. To save those sneaky lt and execution values, we simply select the fields based on their names and extract their values into strings!

Now that we have these sneaky auto-generated values, we can attempt to submit a login attempt as so:

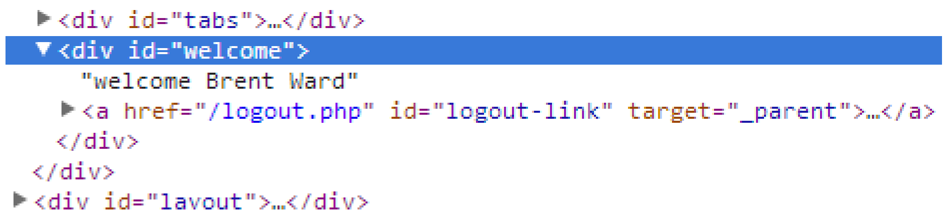
```
doc = Jsoup.connect(mLoginURL)
       .data("username", mUsername)
       .data("password", mPassword)
       .data("lt", ltValue)
       .data("execution", executionValue)
       .data("_eventId", "submit")
       .data("submit", "Log in")
       .cookies(loginForm.cookies())
       .post();
```

Again, the variable mLoginURL is a string variable for the URL of the page we are attempting to login to, in this case Pipeline. Each of the .data segments takes two arguments – the name of the input field, and the data we wish to insert into it. We can see that the "username" field takes the variable mUsername, which is the string we obtained from the user earlier. Likewise, "password" takes mPassword. In the "lt" and "execution" fields, we will submit the variable strings we just saved for those values.

If you remember from the JavaScript Console, the login form also always contained an "_eventId" field and a "submit" field. Since the values in these fields are always the same for any user, we can hardcode the values ourselves to "submit" and "Log in".

Finally, you will notice that we submit the cookies from the loginForm.cookies(). This submits all of the auto-generated cookies created once a user attempts to login, such as a session ID. We don't actually need to analyze the contents of these cookies, nor do we need to understand them – we simply know that they are necessary to login and they are easy to add, so we will add them. Once we have provided input for all of these fields, we simply say ".post()" to submit our login form.

So now that we've done all that, how do we find out if the login was successful or not? It's actually a lot simpler than one would first think, given the difficulty in the previous steps. If you login to Pipeline, you will notice in the top right corner a message that says "welcome" followed by your name. If we examine it with the JavaScript Console, we will see that it is part of a <div> tag. Clicking on the arrow next to the <div> tag in the console will reveal its inner HTML, as seen below:


```
▶<div id="tabs">…</div>
▼<div id="welcome">
   "Welcome Brent Ward"
  ▶<a href="/logout.php" id="logout-link" target="_parent">…</a>
 </div>
</div>
▶<div id="layout">…</div>
```

We can save the text from this <div> tag as a string in just one line of code!

```
mWelcomeMessage = doc.select("div[id=welcome]").html();
```

Now, here’s the interesting part. If the login attempt was successful, the Jsoup document will contain this <div> tag which displays the user’s name. If the login failed, the document would contain the HTML from the login page, since a failed login redirects back to the login page. In that case, the “welcome” <div> tag would not be present.

However, in the line of code above, we attempted to save the text from the welcome <div> tag whether we were successful in logging in or not. This is because, in the event that the login failed and the div tag could not be found, the string we created would simply become an empty string with zero characters. To see if the login was successful or not, we can simply perform an if-else test based on the length of the string we created!

```
if (mWelcomeMessage.length() > 0)
        // This shortens the contents of Pipeline's #welcome div-tag to display only
        // the user’s name
        mWelcomeMessage = mWelcomeMessage.substring(8,mWelcomeMessage.length()-67);
else
        // This displays if the user was unable to log in successfully
        mWelcomeMessage = getString(R.string.login_failed);
```

So, if the login was successful and the <div> tag was found, the length of mWelcomeMessage would be greater than 0. If that is the case, in the code above, my team shortened the string we found into a substring which only contains the user’s real name. Instead of doing that, you may insert any sort of code that you want to handle a successful login.

In the event that we did not find the <div> tag, mWelcomeMessage would be 0 characters long. In the code above, my team decided to change the text to inform the user that the login failed. But again, you may place any code you want to handle a failure within the else statement instead.

Once we have determined whether the login was successful or not, we are finally ready to call the onPostExecute method of our asynchronous task. This will automatically execute after the Jsoup tries to login to your desired website. Depending upon what your app does, the code in this section will vary wildly, so I will not attempt to display example code here. Instead, simply know that you should dismiss your progress dialog from earlier and have two possible outcomes depending on whether the user was able to successfully login or not.

In HU Pal, if the username and password provided were valid, we saved them to the app’s shared preferences file and brought the user to the main screen of our app. If the credentials were invalid, the user would remain on the login page activity and be notified that the login had failed. You may wish to do something different with your app, but once you have validated the credentials as I have outlined above, you are free to program whatever you want!

Login Successful! But, what now?

So now that we have validated the user's credentials, how are we able to navigate around within a website? Jsoup did not actually create and establish a solid connection – instead, it got past the login page, grabbed the text that we needed, and quit. How do we show a page within a web site as if the user had logged in?

Surprisingly, it is much easier to navigate through multiple pages than it is to login! Using the Jsoup code provided above, we were able to validate the username and password of a user. Once we have valid credentials, we can display buttons to the user to navigate to certain pages on a website. For example, in HU Pal, one of the buttons allows the user to navigate through multiple pages to view their chapel skips with a single press.

However, once the Chapel button is pressed, we open a webview which navigates through multiple pages until we arrive where we want the user to be. Until the webview reaches its destination, we don't exactly want the user to see multiple web pages flashing by in succession, so we hide what we are doing behind another progress dialog. See below:

```
pd = new ProgressDialog(getActivity());
pd.setMessage("Getting Chapel Info...");
pd.show();

mWebview = (WebView)v.findViewById(R.id.webView);
mWebview.loadUrl(url);
mWebview.setVisibility(View.INVISIBLE);
mWebview.getSettings().setJavaScriptEnabled(true);
```

Once we display the progress dialog to let the user know there will be a short wait, we create a webview and give it a URL to connect to. In this case, we provide the URL of the chapel info page of Pipeline. We then set the webview temporarily to invisible so the user does not see our background work (which also prevents the user from clicking on the webview when we do not want them to). Finally, we enable javascript on the webview so that we may manipulate it.

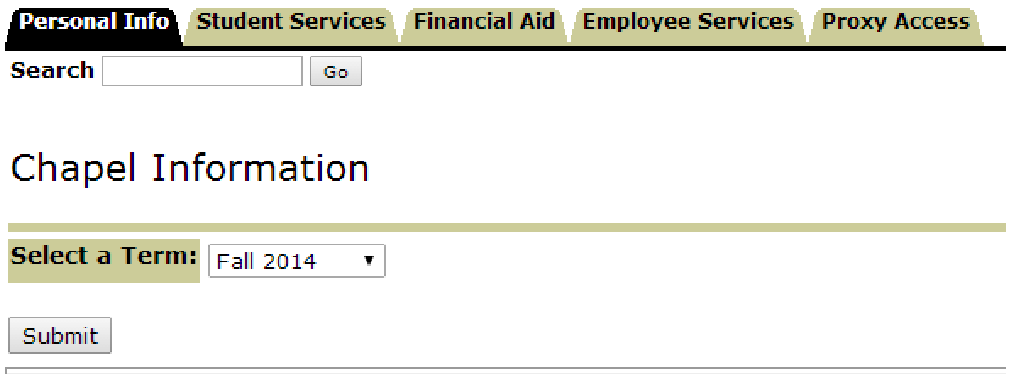
Next, we start navigating through pages on the webview by using javascript. When we launch the webview initially, the user has not yet logged in, so we arrive at the login page. However, we have already confirmed that the user has given valid credentials using Jsoup before, so we may use their username and password (which we saved and retrieved from the shared preferences) to login programmatically. Like so:

```
mWebview.setWebViewClient(new WebViewClient() {
      public void onPageFinished(WebView view, String url) {
            if(mUsername != null || mPassword != null) {
                  mWebview.loadUrl("javascript: {" +
                  "document.getElementById('username').value = '"+mUsername+"';" +
                  "document.getElementById('password').value = '"+mPassword+"';" +
                  "document.getElementsByName('submit')[0].click();" +
                  "};");

                  onPageFinshed2(view, url);
            }
      }
```

This code uses javascript to inject the correct username and password into their respective fields, and then programmatically clicks on the submit button! Once we have done that, we call the onPageFinished2 method, which again attempts to connect our webview to the chapel info page.

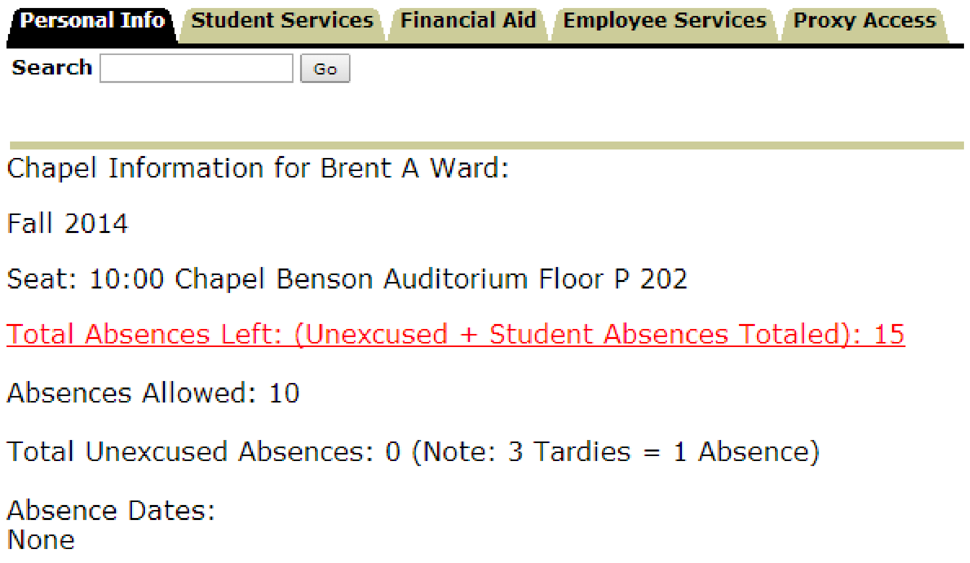
This time, since the user has technically logged in, we arrive here:



To progress, we need to click on the submit button, which will automatically show chapel info for the most current semester. To do this, we use the JavaScript Console to examine the submit button we want to click on. Once we have found its info, we perform a click on it programmatically like so:

```
public void onPageFinshed2(WebView view, String url) {
   mWebview.loadUrl("javascript:document.getElementsByTagName('input')[2].click();");
```

Now that we have clicked on the submit button, the webview should look like this:



Chapel Information for Brent A Ward:

Fall 2014

Seat: 10:00 Chapel Benson Auditorium Floor P 202

Total Absences Left: (Unexcused + Student Absences Totaled): 15

Absences Allowed: 10

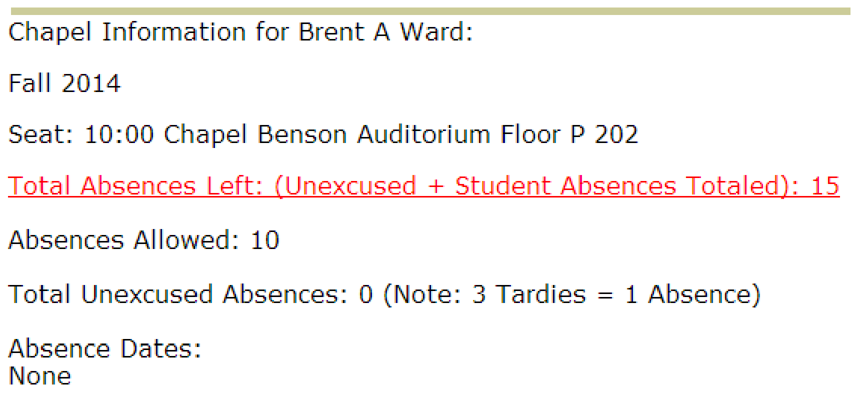
Total Unexcused Absences: 0 (Note: 3 Tardies = 1 Absence)

Absence Dates:
None

Nice! However, there are still some unnecessary tags and links that we want to get rid of. Using the JavaScript Console once again, we can find the names of the areas of the page we wish to hide and set them to be invisible, like so:

```
mWebview.loadUrl("javascript: { " +
"document.getElementsByClassName('headerwrapperdiv')[0].style.display = 'none';" +
"document.getElementsByTagName('center')[0].style.display = 'none';" + "};");
```

Once that code has executed, the webview should look more like this:



Chapel Information for Brent A Ward:

Fall 2014

Seat: 10:00 Chapel Benson Auditorium Floor P 202

Total Absences Left: (Unexcused + Student Absences Totaled): 15

Absences Allowed: 10

Total Unexcused Absences: 0 (Note: 3 Tardies = 1 Absence)

Absence Dates:
None

…which is exactly what we want the user to see! Now that our webview is right where we want it to be, we should let the user see their precious info. Like so:

```
if (pd != null) {
      pd.dismiss();
      mWebview.setVisibility(View.VISIBLE);
}
```

This simply hides the progress dialog which the user has been looking at and makes our invisible webview visible once more. This leaves only one last thing for us to take care of:

```
mWebview.clearCache(true);
mWebview.clearHistory();
CookieSyncManager.createInstance(this.getActivity());
CookieManager cookieManager = CookieManager.getInstance();
cookieManager.removeAllCookie();
```

This clears the cache and history of the webview, as well as the cookies of the present activity. This is important because it ensures no traces of our web activity are left behind. This prevents potential security errors arising from stored usernames and passwords over-staying their welcome for other people to use.

Congratulations!

That's it! If you have followed and understood the tutorial thus far, you have already cleared the steep learning curve of connecting to websites you do not own! It does not take a tremendous amount of code to validate user credentials or to programmatically navigate to a specific webpage. The only tricky part is taking that first step, struggling with a website to see how it works and understanding what it takes to establish a connection.

Although this tutorial focused on connection to Pipeline, the overall methods of connecting to other websites should be similar. The code provided should be close enough to what you will need so that you may copy/paste it for a framework and modify it to suite your needs.