



Getting Started

eBay ActionScript 3.0 Library

Table of Contents

TABLE OF CONTENTS	2
1 WELCOME	4
1.1 ABOUT THE DOCUMENTATION.....	4
1.2 HOW THIS DOCUMENTATION IS ORGANIZED.....	4
1.3 INTENDED AUDIENCE.....	5
1.4 SYSTEM REQUIREMENTS.....	5
2 PRODUCT OVERVIEW	6
2.1 EBAY WEB SERVICE PROTOCOLS.....	6
2.2 SUPPORTED USE CASES.....	6
3 YOUR FIRST API CALL	7
3.1 SETTING UP THE ENVIRONMENT.....	7
3.2 GETTING AUTH'D.....	10
3.2.1 <i>Introducing “Auth n Auth”</i>	10
3.2.2 <i>Setting the values</i>	11
3.2.3 <i>Working in the Production Server</i>	12
3.3 MAKING THE CALL.....	12
3.4 HANDLING THE RESULTS.....	14
4 A MORE ADVANCED EXAMPLE	17
4.1 SETTING UP THE ENVIRONMENT.....	17
4.2 GETTING AUTH'D.....	19
4.3 MAKING THE CALL.....	20
4.4 HANDLING THE RESULTS.....	21
5 API CALL REFERENCE GUIDE	26
5.1 ADDTOWATCHLIST.....	26
5.2 GETITEM.....	26
5.3 GETMYEBAYBUYING.....	26
5.4 GETSEARCHRESULTS.....	27
5.5 GETUSER.....	27
5.6 PLACEOFFER.....	28
5.7 REMOVEFROMWATCHLIST.....	28

6	APPENDICES	29
6.1	ADOBE FLEX REFERENCE MATERIALS	29
6.2	EBAY REFERENCE MATERIALS	29
6.3	OTHER REFERENCE MATERIALS	29

1 Welcome

Welcome to the eBay ActionScript Library! The eBay ActionScript library is a tool that developers can use to build interactive and engaging applications leveraging both eBay's platform and services with the Adobe Flash Player 9 runtime! This library is written in ActionScript 3.0 so any environments using ActionScript 3.0 can use this library, including Adobe Flex 2 and Adobe Flash Professional 9.

1.1 *About the documentation*

This manual is intended as a starter document for developers wanting to use the eBay API with ActionScript 3.0. The actual library is an ActionScript wrapper to the eBay XML API. This document will use Adobe Flex Builder as the primary development environment, providing examples of simple API calls including the necessary code, as well as giving screenshots.

1.2 *How this Documentation is Organized*

This documentation is organized into three major components:

- “[Product Overview](#)” on pg. 5 will give a description of the library, its structure as well as intended and supported use cases.
- “[Your First API Call](#)” on pg. 6 gives you a deeper look at the library. It will describe basic information and instructions on developing and authorizing your first call. Use this section as a guide to making your first API call → `GetUser()`.
- “[A More Advanced Example](#)” on pg. 17 provides a more complex, yet likely a more common, use of the library. Now that you've made your first call, you can replicate most of it here by performing a more common API call → `GetSearchResults()`.
- “[API Call Reference Guide](#)” on pg. 26 lists all of the supported API calls included in this library as well a brief description of usage.

1.3 *Intended Audience*

We assume that you are a developer who is a member of the eBay Developers Program¹ and who has a comfortable working knowledge of web APIs. There is introductory Flex material as well as more advanced techniques, so prior knowledge of Flex is a plus, but not necessary.

1.4 *System Requirements*

Supported Client Development Environments:

- Adobe Flex 2
- Adobe Flash Professional 9²

Library Dependancies³

- CoreLib
- FlexUnit

All of the examples in this document will be given in Adobe Flex 2. You can download a trial version of Flex Builder from <http://www.adobe.com/go/flextrial>.

¹ Join the eBay Developers Program at <http://developer.ebay.com/>

² Not yet released at time of publication; there is a preview available on [Adobe Labs](http://labs.adobe.com/technologies/flash9as3preview/) at <http://labs.adobe.com/technologies/flash9as3preview/>

³ Find libraries at <http://actionscript3libraries.riaforge.org/>

2 Product Overview

Access to eBay Web Services enable third-party developers to create novel and innovative applications leveraging the eBay marketplace in new ways. Many users interact with eBay using the standard web interface; however, use of their Web Services offers third-party application developers the ability to create a wide array of different solutions for over 100 million users.

2.1 eBay Web Service Protocols

eBay's Web Services include support for two major protocols: XML/ HTTPs & SOAP. While ActionScript 3.0 has support for SOAP (via their [SOAP Package](#)), the ActionScript eBay Library actually takes advantage of their XML API. ActionScript 3.0, being an ECMAScript based language, uses a powerful XML handling standard called E4X⁴ (ECMAScript for XML). This standard adds native XML datatypes to the language, adding familiar object-oriented operators to XML-based data allowing for simple, yet fast, handling and manipulation of XML data.

2.2 Supported Use Cases

- Add an item to single user's watch list
- Get information about a single item
- Retrieve all listings watched by a single user
- Retrieve all listings of auction items won by a particular user
- Retrieve all listings of auction items lost by a particular user
- Retrieve all listings of auction items bid on by a particular user
- Get information about a single user
- Submit a bid to a single item
- Remove an item from a single user's watch list

⁴ Read more at <http://www.ecma-international.org/publications/standards/Ecma-357.htm>

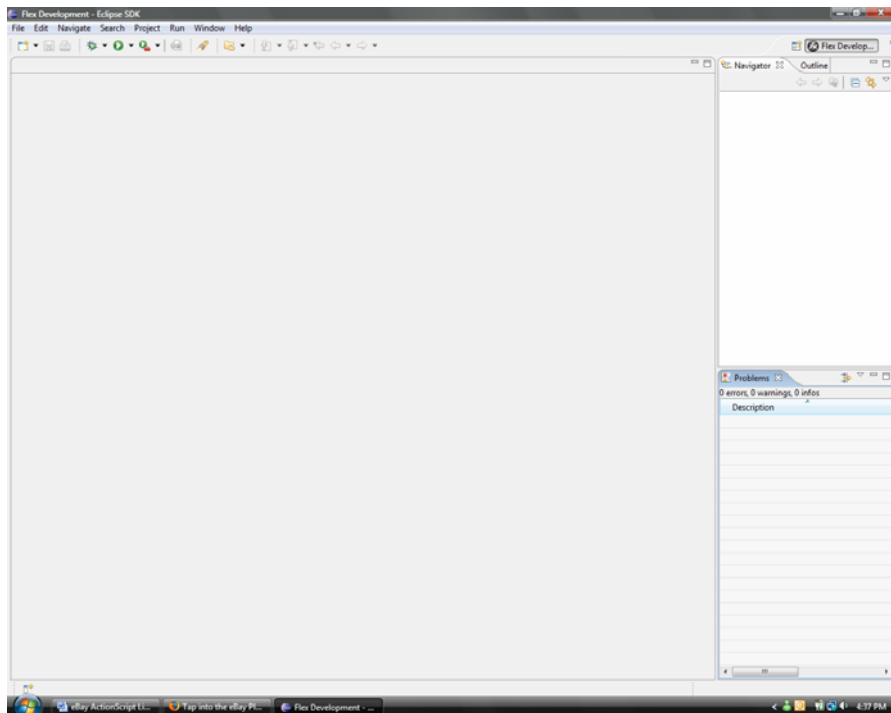
3 Your First API Call

Use of the API through this library has been divided into four main steps: set up the environment, get authenticated, make the call, and handle the results.

3.1 *Setting up the Environment*

Since the examples in this document will be based on applications in Adobe Flex, it will be beneficial to start from the beginning: creating a new Flex project in Flex Builder⁵.

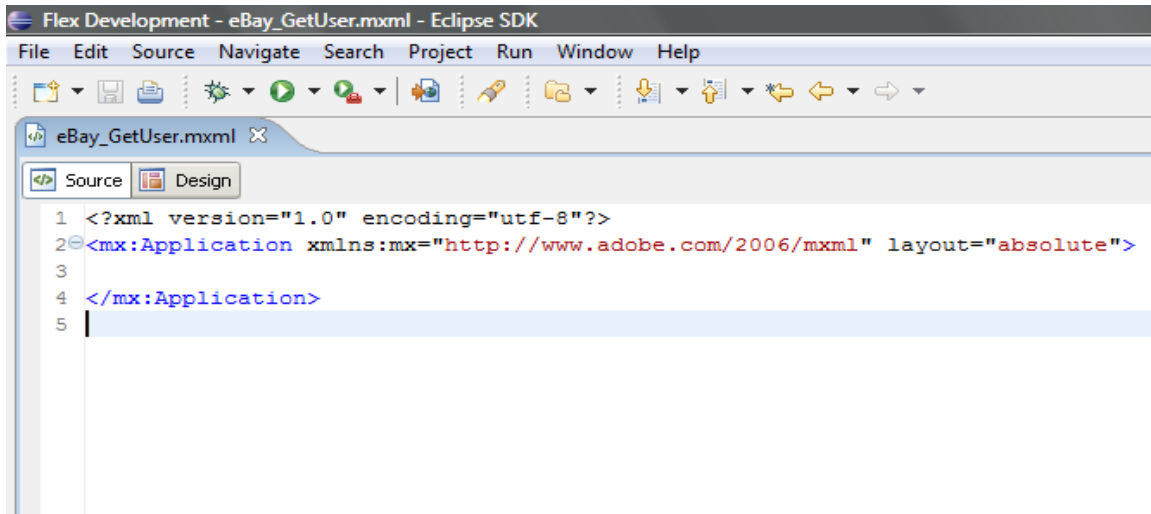
If you download Flex Builder and open up the application, you should get something that looks like...



Flex Builder IDE

To create a new Flex Project, choose “File → New → Flex Project” from the menu at the top. Give your project a name and click *Finish*. You should now have a project in your Navigator, as well as a new .MXML file opened in the application.

⁵ Flex Builder trial version can be downloaded at <http://www.adobe.com/go/openflextrial>



```
1 <?xml version="1.0" encoding="utf-8"?>
2 <mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute">
3
4 </mx:Application>
5 |
```

Standard starting MXML file for a new project

For our sample application, we want to make an API call to eBay's `GetUser` function. The application will allow a user to enter in an eBay user ID, and it will simply return the user's e-mail address. Just to make things a little clearer, we will also output the raw XML result returned from the eBay API call. Now, we will just layout the controls.

We want a simple interface, so we'll only include a text input to enter the user ID, a label to output the e-mail address, and finally a text box to display the full XML raw result returned. We will also add a `<mx:Script>` section, which will contain all of our ActionScript 3.0 logic. And just for the sake of aesthetics, we'll change the layout property in the `<mx:Application>` tag at the top of the page from `"absolute"` to `"vertical"`. We should have something like...

```

<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical">

    <mx:Script>
        <![CDATA[
            // all actionscript data goes here!
        ]]>
    </mx:Script>

    <!-- Text input to enter user ID -->
    <mx:HBox>
        <mx:TextInput id="user_input"/>
        <mx:Button label="Get User"/>
    </mx:HBox>

    <mx:Spacer height="10"/>

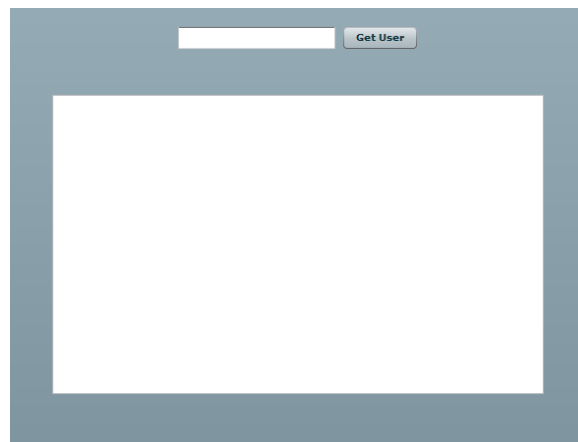
    <!-- Label to display e-mail address -->
    <mx:Label id="email_label"/>

    <!-- Text box to display full XML raw result -->
    <mx:TextArea id="result_box" height="300" width="500"/>

</mx:Application>

```

When we compile and run this application, we will get a very simple .SWF.



UI layout for sample application

Now that we have the layout, we can start getting authenticated.

3.2 *Getting Auth'd*

3.2.1 Introducing “Auth n Auth”

To make a successful API call, you must have a set of developer keys. For every API call sent to the eBay servers, these keys must be sent as well so that eBay can ensure that this particular call is coming from a valid member of the eBay Developers Program.

Developers can get authenticated by eBay via the Authentication and Authorization (Auth n Auth) feature. By obtaining developer keys, Auth n Auth enables the developer to interact with eBay using only the development keys, without ever having to enter their password.

If you haven't done so already, you can join the eBay Developers Program at <http://developer.ebay.com/>. Once you have joined, you can then get your development keys for the sandbox servers⁶ and even the production servers. While development keys are essential in the Auth n Auth process, an important mechanism is the authentication tokens. Once keys are obtained, you can then use those keys to generate authentication tokens⁷. These tokens have a limited lifespan and multiple tokens can be attributed to a single set of developer keys. This means that a single application can have a single developer yet have multiple users.

Before continuing, make sure you have the following authentication items:

- Development Keys (includes DevID, AppID, and CertID)
- Authentication Token (Sandbox, Production, or both)

⁶ All examples in this document will be made using calls to the sandbox.

⁷ Once development keys have been obtained, get authentication tokens here at <http://developer.ebay.com/tokentool/>.

3.2.2 Setting the values

As part of the library there is a dedicated class to authentication, aptly named `AuthAndAuth`⁸. There are static variables within this class which must be set in order to make a successful API call. Simply set them before you send your call, like so...

```
AuthAndAuth.apiDevName = // insert your sandbox DevID here
AuthAndAuth.apiAppName = // insert your sandbox AppID here
AuthAndAuth.apiCertName = // insert your sandbox CertID here
AuthAndAuth.apiAuthToken = // insert your sandbox Auth Token here
```

For the sample application, we must put this in an `ActionScript` function and place that function within the `<mx:Script>` tags. We'll call this function `Init()` and we'll call it when the application finishes creation by setting the `creationComplete` property to `"Init()"` in the `</mx:Application>` at the top.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical"
    creationComplete="Init()">

    <mx:Script>
        <![CDATA[
            import com.adobe.webapis.ebay.*;

            public function Init():void
            {
                AuthAndAuth.apiDevName = // your DevID
                AuthAndAuth.apiAppName = // your AppID
                AuthAndAuth.apiCertName = // your CertID
                AuthAndAuth.apiAuthToken = // your Auth Token
            } // Init
        ]]>
    </mx:Script>

    ...

</mx:Application>
```

We're now authorized. Time to make our API call.

⁸ Located within the library at `com.adobe.webapis.ebay.AuthAndAuth`

3.2.3 Working in the Production Server

In addition to the authentication static variables in `AuthAndAuth`, there is another static variable called `isSandbox`. It is a Boolean and is defaulted to `true`. So unless stated otherwise, all of your calls will be made to the sandbox servers. If you wish to use the production servers, simply set `AuthAndAuth.isSandbox = true`. Note that if you do this, you will have to change your development keys and authentication token to your production ones.

3.3 Making the Call

In order to make an eBay API call, you must first make an `EBayService` object. The `EBayService` class contains the wrapper information for all the available calls, so once you've created this object, you have access to all the available calls.

You can create the object and make the call like so...

```
// create EBayService object
public var eBay:EBayService = new EBayService();

// make the call by calling this function
public function PerformCall():void
{
    eBay.getUser("johndoe");    // replace with your username9
} // PerformCall
```

For the sample application, we would also place this within the `<mx:Script>` section. We will change one thing however. Instead of passing in "johndoe" as the user ID to look up, we will pass in the `text` property of the input box from our UI (which we had named "user_input") that we created in section 3.1. We do this by replacing "johndoe" with `user_input.text`.

⁹ Remember, you are searching the sandbox servers by default, so unless you have created a sandbox user ID in addition to your production user ID, don't expect to find anything associated with your user ID.

```

<mx:Script>
  <![CDATA[
    import com.adobe.webapis.ebay.*;

    public var eBay:EBayService = new EBayService();

    public function Init():void
    {
      AuthAndAuth.apiDevName = // your DevID
      AuthAndAuth.apiAppName = // your AppID
      AuthAndAuth.apiCertName = // your CertID
      AuthAndAuth.apiAuthToken = // your Auth Token
      AuthAndAuth.isSandbox = false;
    } // Init

    public function PerformCall():void
    {
      eBay.getUser(user_input.text);
    } // PerformCall
  ]]>
</mx:Script>

```

Now that we've set up our authentication and created our function to make the API call, we just need to call that function. We will make it so that when the user clicks on the button labelled "Get User", it will call the function and send the API call. We do this by changing the click event handler of the button by adding `click="PerformCall()"`.

```

<!-- Text input to enter user ID -->
<mx:HBox>
  <mx:TextInput id="user_input" enter="PerformCall()"/>
  <mx:Button label="Get User" click="PerformCall()"/>
</mx:HBox>

<mx:Spacer height="10"/>

<!-- Label to display e-mail address -->
<mx:Label id="email_label"/>

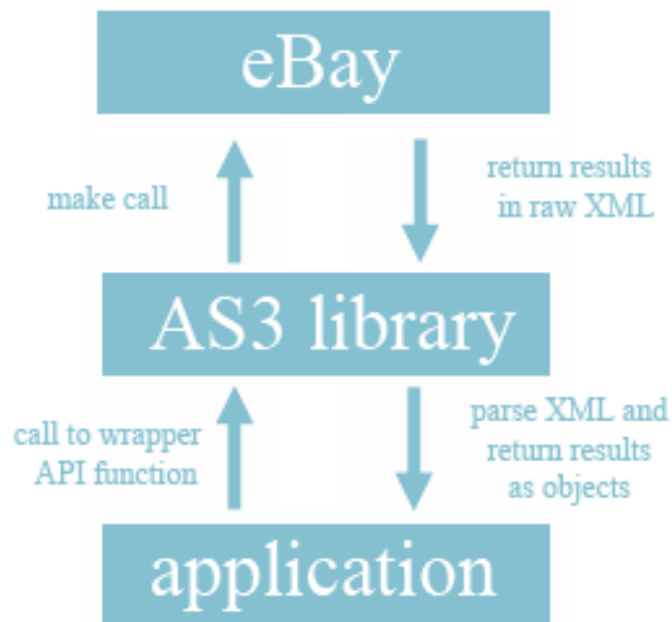
<!-- Text box to display full XML raw result returned from API call -->
<mx:TextArea id="result_box" height="300" width="500"/>

```

We've sent the call, but we are doing nothing with the results. Let's create a result handler.

3.4 Handling the Results

After sending the API call, the results are returned in XML. These results are caught by an event handler within the library. The library then parses them, turns them into valid eBay objects (or eBay errors, if an error has occurred), and sends these results back out for us to interpret. We can access these results by creating a results handler of our own, which will catch the result event sent off by the library. The workflow looks like this...



For this particular API call, `GetUser()`, the library will return to us an `eBayResultEvent` with the data parsed as an `EBayUser` object. So, within our result handler, we must take that result and typecast it accordingly. We can do that with the following line of code...

```
var user:EBayUser = ebay_event.data as EBayUser;
```

Now, our object `user` is a valid `EBayUser` object and has all the properties that an actual eBay user has. One such property is an e-mail address. Since this data is now an

object, you can now access this property as `user.email`. Adding in some validations, we should have a result handler that looks like so...

```
public function GetUserCompleteHandler(ebay_event:EBayResultEvent):void
{
    var user:EBayUser = ebay_event.data as EBayUser;

    if (user.ack == Ack.SUCCESS)
    {
        email_label.text = "e-mail:" + user.email;
        result_box.text = user.rawResult10;
    } // if statement
    else
    {
        email_label.text = user.errors[0].shortMessage;
    } // else statement
} // GetUserCompleteHandler
```

The last thing we need to do for this event handler is to tie this function to a particular event. We do this by using a member function called “`addEventListener`” which will call this particular event handler whenever the results of a `GetUser()` call are returned.

```
ebay.addEventListener(EBayResultEvent.ON_GET_USER,
    GetUserCompleteHandler);
```

This can be placed anywhere within the `Init()` function we created earlier, so as to register this event listener right when we run the application. A final look at our code should give us something similar to...

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical"
    creationComplete="Init()">

<mx:Script>
    <![CDATA[
        import com.adobe.webapis.ebay.*;
        import com.adobe.webapis.ebay.results.*;
        import com.adobe.webapis.ebay.events.*;
        import com.adobe.webapis.ebay.codes.*;
```

¹⁰ `rawResult` is a member of *every* object returned from an API call. This variable is put there for the sole purpose of debugging, in the case the parsing fails and/or relevant error objects aren’t returned.

```

public var eBay:EBayService = new EBayService();

public function Init():void
{
    AuthAndAuth.apiDevName = // your DevID
    AuthAndAuth.apiAppName = // your AppID
    AuthAndAuth.apiCertName = // your CertID
    AuthAndAuth.apiAuthToken = // your Auth Token
    AuthAndAuth.isSandbox = false;

    eBay.addEventListener(EBayResultEvent.ON_GET_USER,
                          GetUserCompleteHandler);
} // Init

public function PerformCall():void
{
    eBay.getUser(user_input.text);
} // PerformCall

public function GetUserCompleteHandler
    (ebay_event:EBayResultEvent):void
{
    var user:EBayUser = ebay_event.data as EBayUser;

    if (user.ack == Ack.SUCCESS)
    {
        email_label.text = "e-mail:" + user.email;
        result_box.text = user.rawResult;
    } // if statement
    else
    {
        email_label.text = user.errors[0].shortMessage;
    } // else statement
} // GetUserCompleteHandler
]]>
</mx:Script>

<!-- Text input to enter user ID -->
<mx:HBox>
    <mx:TextInput id="user_input" enter="PerformCall()"/>
    <mx:Button label="Get User" click="PerformCall()"/>
</mx:HBox>

<mx:Spacer height="10"/>

<!-- Label to display e-mail address -->
<mx:Label id="email_label"/>

<!-- Text box to display full XML raw result returned from API call -->
<mx:TextArea id="result_box" height="300" width="500"/>

</mx:Application>

```

4 A More Advanced Example

Now that you've made your first call, here is a more involved example using a more common API call, `GetSearchResults`. Again, like the first example, we'll set up the environment, get authenticated, make the call, and handle the results. Most of this is very similar to the previous example, so make sure you've read and understand that example before moving on.

4.1 Setting up the Environment

We'll start by creating a new Flex Project. Beginning with the standard MXML file that all Flex projects begin as, we will start to add our interface. Since we are doing a "Search" page, we will want to have a search box, a search button, and a place to list our search results. Since we are using Flex, we can use a built-in component suited for such applications called a `DataGrid`. A `DataGrid` is simply a grid that displays information in a list-view based on a given data source. Finally, just for aesthetics, we will put this `DataGrid` within a `Panel`, to give it a border, and also to display relevant information to us. You'll see later how we can do this, but for now, the code to add the following components should look something like this...

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical">

    <mx:Script>
        <![CDATA[
            // all actionscript data goes here!
        ]]>
    </mx:Script>

    <!-- Text input to enter search query -->
    <mx:HBox>
        <mx:TextInput id="query_input" enter="PerformCall()"/>
        <mx:Button label="Search" click="PerformCall()"/>
    </mx:HBox>

    <mx:Spacer height="30"/>

    <!-- datagrid populated with search results -->
    <mx:Panel
```


4.2 Getting Auth'd

This part is exactly the same as the previous example, using the GetUser call (and should be similar, if not identical, with all the other calls as well). All we must do is set the developer keys and authentication token prior to making the actual API call.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="vertical"
  creationComplete="Init()">

  <mx:Script>
    <![CDATA[
      import com.adobe.webapis.ebay.*;

      public function Init():void
      {
        AuthAndAuth.apiDevName = // your DevID
        AuthAndAuth.apiAppName = // your AppID
        AuthAndAuth.apiCertName = // your CertID
        AuthAndAuth.apiAuthToken = // your Auth Token
      } // Init
    ]]>
  </mx:Script>

  ...

</mx:Application>
```

Time for the API call.

4.3 Making the Call

Again, this section is very similar to the previous example. We create the EBayService object, create a function to make the call, and then call that function.

So, for our example, our `<mx:Script>` tag will now look like...

```
<mx:Script>
  <![CDATA[
    import com.adobe.webapis.ebay.*;
    import com.adobe.webapis.ebay.results.*;
    import com.adobe.webapis.ebay.events.*;
    import com.adobe.webapis.ebay.codes.*;
    import com.adobe.webapis.ebay.objects.*;

    public var eBay:EBayService = new EBayService();

    private function Init():void
    {
        AuthAndAuth.apiDevName = // your DevID
        AuthAndAuth.apiAppName = // your AppID
        AuthAndAuth.apiCertName = // your CertID
        AuthAndAuth.apiAuthToken = // your Auth Token
        AuthAndAuth.isSandbox = false;
    } // Init

    private function PerformCall():void
    {
        eBay.getSearchResults(query_input.text);
        results_panel.status='...loading search results...'
    } // PerformCall
  ]]>
</mx:Script>
```

Notice that we've changed the status property of the results panel '**...loading search results...**'. This is one of the aesthetic touches added to the application through the Panel, mentioned in Section 4.1. Now that we've set up the logic, let's call the function we've just created by the changing the click event handler of the button by adding `click="PerformCall()"`.

```

<!-- Text input to enter search query -->
<mx:HBox>
    <mx:TextInput id="query_input" enter="PerformCall()"/>
    <mx:Button label="Search" click="PerformCall()"/>
</mx:HBox>

<mx:Spacer height="30"/>

<!-- datagrid populated with search results -->
<mx:Panel
    id="results_panel"
    width="100%" height="100%"
    creationCompleteEffect="Fade">
    <mx:DataGrid id="results_grid" width="100%" height="100%">
        <mx:columns>
            <mx:DataGridColumn
                headerText="Title" />
            <mx:DataGridColumn
                headerText="Current Price" width="200"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Panel>

```

Once we've made the call, we need to handle the results.

4.4 Handling the Results

This part of the example is where it gets slightly more involved. Usage of the library has so far remained relatively the same as our first example with the `GetUser` call, but now that we are returning a more complex object, specifically a `SearchResults` object rather than an `EBayUser` object, there are a few details that must be mentioned in handling these results. But, first things first, just as we did before, we must typecast our `eBayResultEvent` data to the appropriate type.

```
var searchResults:SearchResults = ebay_event.data as SearchResults;
```

Now we have our `SearchResults` object, which we've called `searchResults`. Just as the `EBayUser` object, there are public member variables that are available for me to access, some of which are objects themselves. For instance, our `searchResults` object has a member variable called `searchResultItems`, which is actually an array of `items`. So, this member variable contains an array of all of the items that are returned from my query. Here is how to access them.

```

searchResults.searchResultItems;           // returns the array
                                             of items
searchResults.searchResultItems.item[0];   // returns the first
                                             item matching my
                                             query
searchResults.searchResultItems.item[3].itemID; // returns the item ID
                                             of the third item
                                             matching my query

```

With our newly returned results, we would like our DataGrid to be populated with the relevant items, so we need to set the data source of the grid to be the array of items. The DataGrid component has a property called `dataProvider` which will set the data source of the grid. As you may have figured out already, we will set it to `searchResults.searchResultItems`.

```

private function GetSearchResultsCompleteHandler
    (ebay_event:EBayResultEvent):void
{
    var searchResults:SearchResults =
        ebay_event.data as SearchResults;

    if (searchResults.ack == Ack.SUCCESS)
    {
        results_panel.status = "";
        results_grid.dataProvider =
            searchResults.searchResultItems;
    } // if statement
    else
    {
        results_panel.status =
            searchResults.errors[0].shortMessage;
    } // else statement
} // getsearchresults_completehandler

```

Notice that we've also set the results panel's `status` property to "", signifying that we are no longer loading the results since they have just been returned. Our data grid now has its data source set, and we have already determined two properties for it to display: the item's title and the item's current price. For us to complete this, we actually need to create two more functions, *label* functions.

A data grid displays and parses XML data by default. However, we are passing in a fully qualified object, not XML. The data grid does not know how to display the relevant data,

so we must tell it how, by using a label function. A label function is simply a function that takes an object and a grid column, and returns what should be displayed in that column (in our case, the title and current price). Our label functions look like...

```
private function TitleLabel
    (item:Object, column:DataGridColumn):String
{
    var searchResultItem:SearchResultItem = item as SearchResultItem;

    return searchResultItem.item.title + " (" +
        searchResultItem.item.itemID + ")";
} // TitleLabelFunction

private function CurrentPriceLabel
    (item:Object, column:DataGridColumn):String
{
    var searchResultItem:SearchResultItem = item as SearchResultItem;

    return "$" + searchResultItem.item.sellingStatus.currentPrice;
} // CurrentPriceLabelFunction
```

One final step to make our data grid output what we want is to attribute our label functions to the appropriate columns in our grid. We do this by setting the `labelFunction` property in our `<mx:DataGridColumn>`, like so...

```
<!-- datagrid populated with search results -->
<mx:Panel
    id="results_panel"
    width="100%" height="100%"
    creationCompleteEffect="Fade">
    <mx:DataGrid id="results_grid" width="100%" height="100%">
        <mx:columns>
            <mx:DataGridColumn
                headerText="Title"
                labelFunction="TitleLabel"/>
            <mx:DataGridColumn
                headerText="Current Price" width="200"
                labelFunction="CurrentPriceLabel"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Panel>
```

Finally, our last step is to add our event listener...

```
eBay.addEventListener(EBayResultEvent.ON_GET_SEARCH_RESULTS,
    GetSearchResultsCompleteHandler);
```

In the end, our code should look something like...

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="vertical"
  creationComplete="Init()">

<mx:Script>
  <![CDATA[
    import com.adobe.webapis.ebay.*;
    import com.adobe.webapis.ebay.results.*;
    import com.adobe.webapis.ebay.events.*;
    import com.adobe.webapis.ebay.codes.*;
    import com.adobe.webapis.ebay.objects.*;

    public var eBay:EBayService = new EBayService();

    private function Init():void
    {
      AuthAndAuth.apiDevName = // your DevID
      AuthAndAuth.apiAppName = // your AppID
      AuthAndAuth.apiCertName = // your CertID
      AuthAndAuth.apiAuthToken = // your Auth Token
      AuthAndAuth.isSandbox = false;

      eBay.addEventListener
        (EBayResultEvent.ON_GET_SEARCH_RESULTS,
         GetSearchResultsCompleteHandler);
    } // Init

    private function PerformCall():void
    {
      eBay.getSearchResults(query_input.text);
      results_panel.status='...loading search results...'
    } // PerformCall

    private function GetSearchResultsCompleteHandler
      (ebay_event:EBayResultEvent):void
    {
      var searchResults:SearchResults =
        ebay_event.data as SearchResults;

      if (searchResults.ack == Ack.SUCCESS)
      {
        results_panel.status = "";
        results_grid.dataProvider =
          searchResults.searchResultItems;
      } // if statement
      else
      {
        results_panel.status =
          searchResults.errors[0].shortMessage;
      }
    }
  ]]>

```

```

        } // else statement
    } // getsearchresults_completehandler

    private function TitleLabel
        (item:Object, column:DataGridColumn):String
    {
        var searchResultItem:SearchResultItem =
            item as SearchResultItem;

        return searchResultItem.item.title + " (" +
            searchResultItem.item.itemID + ")";
    } // TitleLabelFunction

    private function CurrentPriceLabel
        (item:Object, column:DataGridColumn):String
    {
        var searchResultItem:SearchResultItem =
            item as SearchResultItem;

        return "$" +
            searchResultItem.item.sellingStatus.currentPrice;
    } // CurrentPriceLabelFunction
    ]]>
</mx:Script>

<!-- Text input to enter search query -->
<mx:HBox>
    <mx:TextInput id="query_input" enter="PerformCall()"/>
    <mx:Button label="Search" click="PerformCall()"/>
</mx:HBox>

<mx:Spacer height="30"/>

<!-- datagrid populated with search results -->
<mx:Panel
    id="results_panel"
    width="100%" height="100%"
    creationCompleteEffect="Fade">
    <mx:DataGrid id="results_grid" width="100%" height="100%">
        <mx:columns>
            <mx:DataGridColumn
                headerText="Title"
                labelFunction="TitleLabel"/>
            <mx:DataGridColumn
                headerText="Current Price" width="200"
                labelFunction="CurrentPriceLabel"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Panel>
</mx:Application>

```

5 API Call Reference Guide

5.1 *AddToWatchList*

Use `AddToWatchList` to add a single item to a user's My eBay watch list.

To call this function, make a call to `EBayService.addToWatchList()`, which will create and dispatch an `EBayResultEvent` of type `onAddToWatchList` for the user to catch signifying that the results have been returned from the eBay servers. The result event contains an `AddOrRemoveFromWatchListResult` object, an object containing the results from the `AddToWatchList` API call.

Parameters:

- `itemID:String` – The item ID of the item to be added to the users watch list.

5.2 *GetItem*

Use `GetItem` to retrieve information of a particular item.

To call this function, make a call to `EBayService.getItem()`, which will create and dispatch an `EBayResultEvent` of type `onGetItem` for the user to catch signifying that the results have been returned from the eBay servers. The result event contains an `EBayItem` object, an object containing the information of the specified object.

Parameters:

- `itemID:String` – The item ID of the item for which information to retrieve.

5.3 *GetMyeBayBuying*

Use `GetMyeBayBuying` to retrieve a user's My eBay list, including watch list, bid list, and lost list.

To call this function, make a call to `EBayService.getMyeBayBuying()` which will create and dispatch an `EBayResultEvent` of type `onGetMyeBayBuying` for the user to catch signifying that the results have been returned from the eBay servers. The result event contains a `MyeBayBuyingItems` object, an object containing the `WatchList` items, `LostList` items, and `BidList` items of an authenticated user.

5.4 GetSearchResults

To call this function, make a call to `EBayService.getSearchResults()` which will retrieve search results using the specified query across all categories.

Calling this function will create and dispatch an `EBayResultEvent` of type `onGetSearchResults` for the user to catch signifying that the results have been returned from the eBay servers. The result event contains a `SearchResults` object, an object containing the results from the `GetSearchResults` API call.

Parameters:

- `query:String` – The search query.

5.5 GetUser

Use `GetUser` to retrieve information of the use with the specified user ID.

To call this function, make a call to `EBayService.getUser()` which will create and dispatch an `EBayResultEvent` of type `onGetUser` for the user to catch signifying that the results have been returned from the eBay servers. The result event contains an `EBayUser` object, an object containing the information of the specified user.

Parameters:

- `userID:String` – The user ID of the user whose information to retrieve.

5.6 *PlaceOffer*

User `PlaceOffer` to place a maximum bid on a particular eBay item.

To call this function, make a call to `EBayService.placeOffer()` which will create and dispatch an `EBayResultEvent` of type `onPlaceOffer` for the user to catch signifying that the results have been returned from the eBay servers. The result even contains a `PlaceOfferResult` object, an object containing the results from a `PlaceOffer` API call.

Parameters:

- `itemID:String` – The item ID of the item for which to place the bid.
- `maxBid:String` – The max bidding amount of the bid.

5.7 *RemoveFromWatchList*

Use `RemoveFromWatchList` to remove an single item from a user`s My eBay watch list.

To call this function, make a call to `EBayService.removeFromWatchList()` which will create and dispatch an `EBayResultEvent` of type `onRemoveFromWatchList` for the user to catch signifying that the results have been returned from the eBay servers. The result event contains an `AddOrRemoveFromWatchListResult` object, an object which contains the results from the `RemoveFromWatchList` API call.

6 Appendices

6.1 Adobe Flex Reference Materials

Flex Builder Trial

- <http://www.adobe.com/go/flextrial>

6.2 eBay Reference Materials

eBay Developers Program

- <http://developer.ebay.com/>

eBay XML API Documentation (PDF)

- <http://developer.ebay.com/DevZone/XML/docs/PDF/eBayXMLAPIGuide.pdf>

eBay Sandbox Environment

- <http://sandbox.ebay.com/>

6.3 Other Reference Materials

ActionScript eBay Library – Project Page

- <http://code.google.com/p/actionscript-3-libraries/>

ActionScript eBay Library – Group Page

- <http://groups.google.com/group/as3libraries>

ActionScript eBay Library – Documentation

- <http://actionscript-3-libraries.googlecode.com/svn/trunk/docs/ebay/index.html>

E4X on ECMA-International

- <http://www.ecma-international.org/publications/standards/Ecma-357.htm>