

Develop apps with Web services and the eBay SDK, Part 3: Develop eBay applications with PHP5 and Web services

Skill Level: Intermediate

[Alex Garrett \(ljagged@thinkpig.org\)](mailto:ljagged@thinkpig.org)
Senior Consultant
Isthmus Group, Inc.

08 Nov 2005

Create applications in PHP5 that interact with eBay through Web services. Almost half of eBay's transactions occur through its Web services platform. In this tutorial, you'll acquire a solid understanding of the mechanics of the eBay XML API and learn how to use the `Services_Ebay` PHP extension.

Section 1. Before you start

In June 2004, O'Reilly published a white paper analyzing eBay's Web service deployment. The paper stated that eBay was serving approximately one billion requests each month through its Web service API and that this figure had grown by more than an order of magnitude since 2002. Additionally, one year ago, 40% of all listings on eBay were accomplished through the company's Web services.

I don't know the current figures, but I do know that the developer program is growing by leaps and bounds. eBay continues to refine, adapt, and extend the services it offers through its API. eBay has exposed its entire business model through its API in the hopes that developers will use it to create the next breed of e-commerce applications. This has led to a rich and complex API that requires a fair bit of savvy to master. But eBay works hard to give third-party developers the tools they need to build commercial software offerings.

About this tutorial

This tutorial focuses on developing eBay applications using the XML API with PHP5. The emphasis is on the interface between the application and eBay's Web services, because developing a complete application would be beyond the scope of this tutorial. The tutorial discusses the structure of the request and response XML and the transport mechanism for delivering the requests and retrieving the responses from eBay. It also illustrates in detail how to use the `Services_Ebay` PHP framework.

The techniques discussed fit equally well in a stand-alone application or a Web application. That may seem strange in a PHP tutorial, because PHP is almost always used for Web applications. However, PHP5 has made inroads toward becoming a general scripting language as well as a Web development language. Developing large Web-based e-commerce applications with PHP offers considerable benefit, but there's also merit in developing small, personal scripts that can run outside a Web server. For example, you might write a script that polls eBay, retrieves all your current auctions, gets the current maximum bid for each one, and e-mails you the total every morning. If this were an Amazon Web services tutorial, the obvious next step would be to have the script automatically order the top items on your wish list, up to the value of the total.

Prerequisites

This tutorial assumes that you're comfortable with programming in PHP and have at least a basic understanding of the object model in PHP5. If you need an introduction to object-oriented (OO) programming in PHP, some resources are listed in the [Resources](#) section. Don't be intimidated if you don't have strong OO skills. Although `Services_Ebay` is built in a very strong OO way, using it doesn't require a lot of OO knowledge.

You should also have a basic understanding of XML; you'll be wrangling a lot of it in this tutorial. You should understand the criteria for a well-formed XML document, be comfortable with the terminology used to describe XML, and so on. As before, check the [Resources](#) section if you need a refresher.

Part 2 of the eBay tutorial series explains [how to build a book-distribution application with the eBay Java SDK](#). I highly recommend that you read at least the first half of this tutorial, which provides detailed information about the eBay authentication and authorization mechanism. I'm going to assume you have that knowledge for this tutorial.

In order to execute the examples, you need to have the following software installed:

- PHP5
- CURL
- OpenSSL

Depending on how you built your PHP, you might already have CURL and OpenSSL installed. The easiest way to check is to look at the results of the `phpinfo()` function:

```
$ php -r 'phpinfo();' | grep CURL
CURL support => enabled
CURL Information => libcurl/7.13.1 OpenSSL/0.9.7e zlib/1.2.2
```

Or, you can create a file in a directory accessible to your Web server with the following code:

```
<?php phpinfo(); ?>
```

Most of the examples have been developed to run in PHP's CLI mode (as opposed to CGI). This means execution results are echoed to standard out as text instead of being wrapped up in HTML. The benefit of this approach is that you can put the example in a directory and execute it by typing the following:

```
php -f <filename>
```

You don't need a running Web server. Running the examples in CGI mode is simply a matter of wrapping the output in HTML. For the examples that display XML, you'll also need to escape the output using the `htmlspecialchars` function. This is discussed in the appropriate section.

The final prerequisite for this tutorial is a pretty good understanding of the basic patterns of working with the eBay Web services, especially registering your application with eBay and using the authentication and authorization mechanisms. This subject is discussed at length in the [Java SDK tutorial](#). The application registration and authentication and authorization process is platform agnostic. Rather than cover the same material in this tutorial, I recommend that you read the Java tutorial. The first half will make sense to someone with experience in any programming language. The second half discusses implementing Librivore in Java. It may be helpful to skim those sections, if only to see how much simpler PHP users have it.

If you do nothing else, however, at least use the eBay Token Generation Tool (see [Resources](#)) to generate a token you can use to execute the code in the examples.

Why PHP?

Telling you why you should use PHP is probably unnecessary. Nonetheless, here are a few reasons why using PHP for developing eBay applications is a Good Thing:

- **PHP is good for rapid application development.** It's easy to prototype and incrementally develop applications in PHP. Many of the shortcomings of Java for Rational® Application Developer, such as the static typing and the verbosity of the language, don't exist in PHP.
- **PHP has an enormous number of libraries and extensions.** Hundreds of functions are built into the language for developers to use. And the PHP Extension and Application Repository (PEAR), while not CPAN, has lots of extensions that developers can use instead of reimplementing common functionality.
- **PHP is very well documented.** The PHP documentation really shines. The manual is structured so that developers can comment on the individual sections with clarifications and code examples. Additionally, many good PHP books let developers get a good understanding of PHP easily. I've listed some of my favorites in the [Resources](#) section.
- **PHP is mature.** It's been around since 1994, and it's been in constant development and refinement.
- **PHP scales.** PHP means **Personal Home Page**. It was originally designed by Rasmus Lerdorf as a Perl script to make simple dynamic home pages. Now, PHP is being used to build high-availability distributed applications. IBM has partnered with Zend Technologies to build enterprise applications using PHP and Derby or DB2® UDB.

And there's a subjective reason why you should use PHP: I enjoy programming in PHP. If you're reading this tutorial, I'm guessing that you do, too. I hope this tutorial expands your knowledge of the things you can do with PHP and is enjoyable, as well.

Section 2. eBay XML applications: from first principles

You'll create a basic application from first principles that interacts with eBay. This means you aren't going to use any special tools or frameworks that have been developed for eBay or PHP. Nor are you going to use the code samples from the PHP developer site. I want you to understand everything that's happening across the

wire so that when you use frameworks that abstract away the details, you'll still be comfortable digging into them when necessary.

The eBay request/response cycle

An application that uses the XML API directly goes through the following steps for every API call:

1. The application assembles an XML document that tells eBay what the app wants it to do.
2. eBay parses the XML, executes the action, and returns results as an XML file (provided everything goes according to plan).
3. The application parses the results and then does whatever post-processing is appropriate.
4. In the event that things don't go as planned, the application must parse the return XML for errors and also post-process.

You'll begin by writing a small application that retrieves the details of a user from eBay.

Using the eBay XML API test tool: the request

The test tool is an excellent way to figure out what the XML request/response will look like in your application. It lets you send an XML request with custom parameters and get the response XML.

Go to <http://developer.ebay.com/DevZone/build-test/test-tool.asp>, and set the following:

1. Host: **Sandbox**
2. Certificate: Your developer info
3. User & password: Don't change
4. RequestToken: The token that you created for the application
5. RequestTemplate: **GetUser**

The text box below the RequestTemplate drop-down is populated with an XML

document that represents the request for the `GetUser` API call. Let's look at it:

```
<?xml version="1.0" encoding="utf-8"?>
<request>
<RequestToken>Token</RequestToken>
<DetailLevel>0</DetailLevel>
<ErrorLevel>1</ErrorLevel>
<SiteId>0</SiteId>
<Verb>GetUser</Verb>
<UserId></UserId>
</request>
```

The root element is `<request>`; this is the root element for all API requests. It doesn't have any attributes. Notice that it's all lowercase. Most elements use Pascal case -- the words have initial caps. `request` is one of the few that don't.

The `<RequestToken>` element is used for authentication. All requests require a `RequestToken` with the exception of the `RetrieveToken` API call (for obvious reasons).

`DetailLevel` takes a number indicating how much detail you want eBay to return. The level depends on the particular call. Some calls don't have multiple levels of detail, in which case the value must be 0. Others have a wide variety. The individual calls describe the detail levels. In the case of `GetUser`, a value of 0 returns all of what you want, so use that.

The `ErrorLevel` controls the amount of error reporting. A value of 0 indicates that short error strings should be returned; 1 indicates that long error strings should be returned. During development, you may wish to have long errors returned to help with the debugging process; but in production, you may want to switch to short errors to reduce transmission time.

The `SiteId` is the eBay site against which you're executing the call. For this example, use site ID 0, which is the site in the United States. The documentation has a list of valid site IDs:
http://developer.ebay.com/DevZone/docs/API_Doc/Functions/Tables/SiteIdTable.htm.

The `<Verb>` element is the name of the function you're executing. All of the previous elements apply, regardless of the API call, and need to be present -- they provide information that's not specific to a particular call. In this case, you're invoking the `GetUser` function, so you set the `Verb` element to have a text value of "GetUser".

Each function takes zero or more parameters, which are indicated as elements of the XML. In the case of `GetUser`, it has one required parameter: the ID of the user you're trying to retrieve. You set this parameter by adding a `<UserId>` element to the request. Notice that the `<UserId>` element isn't a child of the `<Verb>` element. In fact, the request XML is almost entirely flat: it consists of a `<request>` element

and then an arbitrary number of child elements that provide the details of the call. The request never goes any deeper than one level below the root node. It has the benefit of making the XML generation fairly easy.

Using the eBay XML API test tool: the response

If you click the Submit button to the right of the drop-down, the API call shown in [Listing 1](#) is executed in the sandbox and the results are returned.

Listing 1. GetUser response XML

```
<?xml version="1.0" encoding="utf-8"?>
<eBay>
  <EBayTime>2005-08-10 22:38:49</EBayTime>
  <User>
    <UserId><![CDATA[librivore]]></UserId>
    <Status>1</Status>
    <SellerLevel>0</SellerLevel>
    <AboutMe>0</AboutMe>
    <RegDate>1995-01-01 11:59:59</RegDate>
    <UserIdLastChanged>2005-06-25 18:30:02</UserIdLastChanged>
    <SiteId>0</SiteId> <IDVerified>1</IDVerified>
    <Star>0</Star>
    <AllowPaymentEdit>1</AllowPaymentEdit>
    <CIPBankAccountStored>0</CIPBankAccountStored>
    <StoreOwner>0</StoreOwner>
    <CheckoutEnabled>1</CheckoutEnabled>
    <MerchandisingPref>1</MerchandisingPref>
    <eBayGoodStanding>1</eBayGoodStanding>
    <IsLAAuthorized>0</IsLAAuthorized>
    <NewUser>0</NewUser>
    <UserIdChanged>0</UserIdChanged>
    <Feedback>
      <Score>0</Score>
      <UniqueNegativeFeedbackCount>-1</UniqueNegativeFeedbackCount>
      <UniquePositiveFeedbackCount>-1</UniquePositiveFeedbackCount>
      <PositiveFeedbackRate>0.0</PositiveFeedbackRate>
    </Feedback>
    <EIAS>nY+sHZ2PrBmdjY+lCpWGpQidj6x9nY+seQ==</EIAS>
    <PaymentType/>
    <SellerGuaranteeLevel>0</SellerGuaranteeLevel>
    <Email><![CDATA[librivore@thinkpig.org]]></Email>
  </User>
</eBay>
```

This is a lot of detail, and the meaning is either obvious or irrelevant for this example. I want to mention a few things, though. First, every result contains the current eBay time; you might find this helpful. Second, notice that this XML isn't flat in the same way the request XML is. It's much more hierarchical. Third, the data is encoded as UTF-8. It's a requirement that data is sent as UTF-8 and the results are returned as UTF-8. If you're working with a different encoding, you'll have to translate to and from UTF-8. I'll discuss this in greater detail later in the tutorial.

Do it in PHP: the Web page

Now that you have an idea of what eBay expects from you and what it will give back, it's time to start your own implementation of the test tool in PHP. Doing so is straightforward, but there are a few tricky bits. Let's start with the page that will be displayed (see [Listing 2](#)).

Listing 2. testtool.php

```
<html><head><title>PHP eBay API Test Tool</title></head>
<body>
<?php
    require_once('ebayTestTool.php');
    require_once('config.php');

    if(isset($_POST['submit'])) {
        $xml = $_POST['xml'];
        $params = array('DEV_ID' => $devId,
            'APP_ID' => $appId, 'CERT_ID' => $certId);
        $results = makeApiCall($xml, $params, $token);
        echo "<pre>";
        echo htmlentities($results);
        echo "</pre><hr/>";
    }

    ?>
    Enter your XML here: <br/>
<form method="POST">
    <textarea name="xml" cols="60" rows="20"></textarea>
    <br/>
    <input type="submit" name="submit" />
</form>
</body>
</html>
```

This is the entry point to the PHP API test tool. The `config.php` file contains the authentication information for the calls. The `ebayTestTool.php` file contains the PHP code that executes the API call. The bottom half of this listing contains the HTML for the form. The `action` attribute is left out of the `form` element because the data in the form should be posted right back to this URL.

The top half of the listing checks to see whether there is any post data. If there is, it arranges the parameters, invokes the `makeApiCall` function, and displays the results. Because the `makeApiCall` function returns an XML document, all of its XML entities must be converted. If you didn't run the results through the `htmlentities` function, the browser would treat it like HTML and display the text elements and not the XML elements.

Do it in PHP: the API invoker

There are three logical parts to making the API call:

- Creating the XML
- Creating the HTTP headers
- Establishing the network connection

There's not much to do for the first item, because the XML is supplied as a parameter to this function; but some changes need to happen before it can be sent to eBay. eBay requires all XML to be encoded as UTF-8, as opposed to ISO-8859-1 or US-ASCII, so you need to translate it and create a header that tells eBay that the XML is encoded properly.

One of the HTTP headers indicates the detail level for the call. Because this information is contained in the XML request, you need to extract that value and store it in the header.

Finally, you check to see whether the `<RequestToken>` element has a value. If it doesn't, you'll use the value of the `$token` parameter. Doing so lets you use the test tool with a sane default but allows you to override it when necessary. [Listing 3](#) shows the `makeApiCall` function.

Listing 3. `makeApiCall` function

```
function makeApiCall($xmlRequest, $params, $token) {
    // add XML PI & convert to UTF-8
    $xmlRequest = "<?xml version=\"1.0\" encoding=\"utf-8\"?>\n"
        .utf8_encode($xmlRequest);
    $callXml = simplexml_load_string($xmlRequest);

    // extract detail level
    $detailLevel = $callXml->DetailLevel;

    // populate Request Token if not populated
    if (strlen($callXml->RequestToken) == 0) {
        $callXml->RequestToken = $token;
        $xmlRequest = $callXml->asXml();
    }

    // create headers
    $headers = createHeaders($callXml->Verb,
        $params['DEV_ID'],
        $params['APP_ID'],
        $params['CERT_ID'],
        $detailLevel);

    // make request
    $responseXml = makeRequest($headers, $xmlRequest);
    return $responseXml;
}
```

The first thing you do is add the XML declaration to the top of `$xmlRequest`. This is a required declaration for any XML that goes to eBay, because it indicates that the

XML has been encoded as UTF-8. XML that either is missing this declaration or has a different encoding will be rejected (see [Listing 4](#)). This function assumes that the `$xmlRequest` doesn't already include an XML declaration. The best way to handle this would be to check for a declaration, see whether it's already encoded as UTF-8, and, if not, adjust the declaration and reencode as UTF-8. However, for the sake of simplicity, you should assume the input doesn't have the declaration and add one.

Listing 4. Error listing

```
<?xml version="1.0" encoding="iso-8859-1" ?><eBay>
<EBayTime>2005-08-12 15:47:20</EBayTime><Errors>
<Error>
<Code>20400</Code>
<SeverityCode>1</SeverityCode>
<Severity>SeriousError</Severity>
<Line>0</Line>
<Column>0</Column>
<ErrorClass>RequestError</ErrorClass>
<ShortMessage>
<![CDATA[Invalid request encoding.]]>
</ShortMessage>
<LongMessage>
<![CDATA[Invalid request encoding. Please use utf-8 encoding
and eliminate any non utf-8 encoding in request content.]]>
</LongMessage>
</Error>
</Errors>
</eBay>
```

The next thing to do is to encode the data as UTF-8. PHP has a built-in function, `utf8_encode`, that takes care of this for you. PHP represents XML data internally as UTF-8, regardless of its original encoding. The conversion is transparent, but you should be aware that if you turn the XML back into a string, you won't get it back with its original encoding. You can find more information on this topic in the [Resources](#) section.

At this point, the XML is in the format that eBay requires; but you still need to populate the `<RequestToken>` element if it doesn't have a value, and you need to extract the value of the `<DetailLevel>` element for the HTTP headers. The easiest way is to use the SimpleXML extension. The `simplexml_load_string` function takes a string representation of an XML document and returns a `SimpleXMLElement` object. This object represents the root of the XML document. You can access and modify the elements of the XML document using the object attribute syntax.

Do it in PHP: setting the headers

It's not enough to simply send the XML document to the eBay server. You also need to set a number of eBay-specific headers that provide the server with information about the call (see [Listing 5](#)). In particular, the headers are where you set your

application developer credentials. This is how eBay validates that you have the right to make the call (the headers also tell eBay whom to bill for API calls).

Creating the headers is a matter of populating an array with the names of the headers followed by the appropriate values.

The eBay XML test tool allows you to set the compatibility level, but here it's hard-coded to 353. The site ID is also hard-coded to 0, the U.S. site. Notice that the X-EBAY-API-SESSION-CERTIFICATE header is made up of the developer credentials separated by semicolons.

Listing 5. createHeaders function

```
function createHeaders ($callName, $devId, $appId,
    $certId, $detailLevel){
    $compatibilityLevel = 353;
    $siteID = 0;

    return array (
        "X-EBAY-API-COMPATIBILITY-LEVEL: ".$compatibilityLevel,
        "X-EBAY-API-SESSION-CERTIFICATE: ".$devId.";".$appId.";".$certId,
        "X-EBAY-API-DEV-NAME: ".$devId,
        "X-EBAY-API-APP-NAME: ".$appId,
        "X-EBAY-API-CERT-NAME: ".$certId,
        "X-EBAY-API-CALL-NAME: ".$callName,
        "X-EBAY-API-SITEID: ".$siteID,
        "X-EBAY-API-DETAIL-LEVEL: ".$detailLevel
    );
}
```

Do it in PHP: sending it across the wire

The last thing to do is to send your data across the wire to the eBay gateway and return the result. The best way to do this is with the CURL extension.

You need to send the data to eBay as a POST request using SSL, and you need to return the results. The only non-obvious parts in [Listing 6](#) are the VERIFYHOST and VERIFYPEER options. Essentially, by setting both of these to false, you tell CURL not to worry about authentication -- just encrypt the data.

Listing 6. makeRequest function

```
function makeRequest ($httpHeaders, $httpBody) {
    $gatewayUrl = 'https://api.sandbox.ebay.com/ws/api.dll';

    $ch = curl_init();
    $res= curl_setopt ($ch, CURLOPT_URL, $gatewayUrl);

    curl_setopt ($ch, CURLOPT_SSL_VERIFYHOST, 0);
    curl_setopt ($ch, CURLOPT_SSL_VERIFYPEER, 0);
}
```

```
curl_setopt($ch, CURLOPT_HTTPHEADER, $httpHeaders);
curl_setopt($ch, CURLOPT_POST, 1);
curl_setopt($ch, CURLOPT_POSTFIELDS, $httpBody);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);

$httpResponse = curl_exec($ch);
curl_close($ch);
return $httpResponse;
}
```

The value of the `$httpResponse` is the XML response you saw earlier with the official eBay API test tool.

Summary

At this point, your test tool should work essentially the same as the official eBay test tool. It doesn't prepopulate the request window with XML templates or return the results in a different window, but it does the job. The best thing (in my opinion) is that it doesn't require Microsoft® Internet Explorer® on Windows® to use!

Feel free to take a break from the tutorial to experiment with the test tool, yours or eBay's. Try some different calls and see what kind of results you get. Try entering some bad data and see how eBay responds.

You should now understand how to:

- Create the request XML with the appropriate `<Verb>` and parameters
- Set the headers
- POST the data to the eBay gateway over SSL using CURL.

And you should have a good understanding of how the eBay XML API works at a low level.

Writing your own test tool wasn't hard, but that's because you weren't doing anything complex. Most of the things you'll be doing with the eBay API are difficult at this level. What you really want to do is work at a higher level of abstraction. That's the subject of the next section.

Section 3. Using Services_Ebay

All of those XML stunts are fine if you're testing the API, but they're a pain if you're trying to write a real application. What you'd rather do is something like the following:

```
<?php
$user = $ebay->GetUser("librivore");
echo "{$user->UserId}'s email address: {$user->Email}\n";
?>
```

Services_Ebay is a framework that allows you to write this kind of code.

Introducing Services_Ebay

Most of the API calls you make in an application are more complex than just retrieving a user. For example, you'll probably want to retrieve all the items a particular user is selling. [Listing 7](#) shows an example of what the code might look like.

Listing 7. Extended GetUser call

```
$user = $ebay->GetUser("librivore");
$send_time_from = gmdate("Y-m-d H:i:s", time()); // now
// one week from now
$send_time_to = gmdate("Y-m-d H:i:s", time() + (60 * 60 * 24 * 7));

$callParms = array( 'DetailLevel' => 32,
                   'PageNumber' => 1,
                   'ItemsPerPage' => 20,
                   'EndTimeFrom' => $send_time_from,
                   'EndTimeTo' => $send_time_to
                 );
$itemsSelling = $user->GetSellerList($callParms);

echo "{$user->UserId} is selling the following:\n";
foreach ($itemsSelling as $item) {
    echo "\tItem number: {$item->Id}\tItem title: {$item->Title}\n";
}
```

As an exercise, you might try implementing that use case using the XML API directly.

These code snippets use the Services_Ebay extension. Services_Ebay is the most fully developed PHP framework for using the eBay XML API. It's also a project in the eBay Community Codebase and has Adam Trachtenberg as one of the project owners. Adam is the technical evangelist for eBay and the respected author of two PHP books. This means using Services_Ebay is a good bet. Much of the rest of this tutorial looks at it in detail.

Most of the details have been encapsulated in a natural OO representation. Rather than wrangle with XML, you tell your `$ebay` object to retrieve the user specified by a particular user ID. Once you have the `user` object, you query it for the items the user is selling. Even though this is a separate eBay API call, it appears to be a standard method invocation.

Unfortunately, you can't wave away all the details. eBay API calls tend to take a lot of parameters, which need to be specified explicitly. The `GetSellerList` API call with a detail level of 32 (which you need to get back the title, among other things) requires that pagination and time filter parameters be supplied.

Installing Services_Ebay

Before you can begin developing with `Services_Ebay`, you need to install it. The easiest way to do so is through PEAR. Execute the following at a command line:

```
pear upgrade -a Services_Ebay-alpha
```

The `-a` option installs all the dependencies that `Services_Ebay` has, and the `-alpha` suffix overrides PEAR's default behavior of not installing unstable packages. At the time of this writing, `Services_Ebay` is still in alpha. If you install `Services_Ebay` in a nonstandard location, be sure the directory is in your include path.

Getting started with Services_Ebay

Let's go back to the original example of retrieving a user from eBay. [Listing 8](#) shows how to do it with `Services_Ebay`.

Listing 8. GetUser with Services_Ebay

```
<?php
    require_once 'config.php';
    require_once 'Services/Ebay.php';

    // pass authentication data to Services_Ebay
    $session = Services_Ebay::getSession($devId, $appId, $certId);

    $session->setToken($token);
    $ebay = new Services_Ebay($session);
    $user = $ebay->GetUser("librivore");
    echo "{$user->UserId}'s email address: {$user->Email}\n";
?>
```

The `config.php` file contains all of your authentication information, just like in the test

tool example. The `Services/Ebay.php` file is the entry point to the framework. First you need to get a `Session`, passing in the developer credentials. Then, you must set the authorization token before you can make any API calls. The developer credentials won't change during the lifetime of the session, but the token might -- especially if you're executing calls on behalf of multiple users. That's why the developer credentials are passed into the `Session` constructor, but the token is set using a method call.

The next step is to create a `Services/Ebay` object. All eBay API calls can go through this object. (You'll see why I say "can" instead of "do" shortly). You know from previous experience with the API that you used a function called `GetUser` as the `<Verb>`, and it took a `UserId` as a parameter. The `Ebay` class also has a `GetUser` method that takes the `UserId` as a parameter and returns a `User` model object.

The `User` model object is a representation of the XML that eBay returns. If you look at the XML from your test tool, you'll see that it has an element called `Email`. You can get this value from the model object by treating it as if it were an attribute of the object and using the arrow operator.

Services_Ebay overview

Now that you've seen an example of how to execute a call with `Services_Ebay`, I'll provide some general background about `Services_Ebay`. `Services_Ebay` is a framework that allows developers to interact with eBay's services in an object-oriented fashion. It consists primarily of the following pieces:

- **`Services_Ebay`** . In addition to being the name of the framework, it's also the name of the main class. It's responsible for acting as a factory for sessions, calls, and models. Calls are invoked through it, and it has hooks for allowing you to override model objects.
- **`Services_Ebay_Session`** . This class is responsible for generating the request XML and unmarshalling the response XML into model data. The session class also parses and exports any errors from eBay and delegates the data transmission to the appropriate `Transport` class.
- **`Services_Ebay_Transport_Curl`** . This class is responsible for taking the data generated by the `Services_Ebay_Session` and sending it to the eBay gateway. It also accepts the response and passes it back to the session. It uses the CURL extension to make the connection. There are also two other `Transport` classes: one uses PHP's `HTTP_Request` object, and the other uses PHP streams. However, there are no provisions for changing the transport driver in the `Session` class.

- **Services_Ebay_Call.** This abstract class is the superclass for all of the API call classes. There is a call class for each eBay *verb* that `Services_Ebay` implements. Each call class is responsible for taking the parameters required for that particular call and invoking the session's `sendRequest()` method with the value of its verb and its parameters.
- **Services_Ebay_Model.** Like `Services_Ebay_Call`, this is an abstract class. It's responsible for creating an object-oriented representation of the entities in eBay's domain. For example, there's a class that represents a `User` that encapsulates attributes like the user ID and the user's e-mail address.

If you were to trace the `$ebay->getUser('librivore')` method call from the previous example, it would look like this:

1. `Services_Ebay` (that is, `$ebay`) loads the `Services_Ebay_Call_GetUser` class.
2. It calls its constructor, passing in the parameter `librivore`.
3. The `GetUser` object checks to ensure the number of parameters is correct and sets the `UserId` attribute to `librivore`.
4. The `Services_Ebay` instance invokes the `call()` method on the `GetUser` object with the session as a parameter.
5. The `GetUser` object invokes the `sendRequest` method on the session with its verb and attributes as arguments (`GetUser` and an array -- `'UserId' => 'librivore'`).
6. The session generates the XML and headers from the parameters and constructs a request much like the eBay test tool.
7. The session loads the `Transport` class and hands the request XML and headers off to it.
8. The transport class makes the connection over HTTPS and sends the data to the eBay gateway.
9. The transport class retrieves the data from the eBay gateway and passes it to the session.
10. The session unmarshalls the response XML into an array and returns the array to the `GetUser` object.
11. The `GetUser` object extracts the `User` data from the array and invokes

the `loadModel()` method on the `Services_Ebay` class, passing in `User` and the user data as parameters.

12. The `Services_Ebay` class locates the `User` model and constructs it with the user data from the call. It then returns the initialized `User` object to the `GetUser` call object.
13. The `GetUser` object returns the `User` model object to the code that invoked `$ebay->GetUser('librivore')` in the first place.

Don't worry if you don't understand all the details. You should understand the responsibilities of the main class types and the how they interact at a high level.

Summary

You should now have a good understanding of the principles behind `Services_Ebay`. It doesn't do anything you couldn't do by manipulating the XML directly, but it makes the process much simpler and object-oriented.

The next section looks at the details of using `Services_Ebay` for authentication, listing items, and searching eBay.

Section 4. Using `Services_Ebay`

This section takes a detailed look at how to invoke the eBay API through `Services_Ebay`. As mentioned in the introduction, Librivore works with multiple users to list the books they want to sell and to provide reporting information on the sales of their books. In order to provide that functionality, you must be able to authenticate users and execute the `AddItem` API call. I'll show you how to implement these using `Services_Ebay`, but I'll show you how to extract the reporting information from the raw XML. This will give you the skills you need to work at whatever level of abstraction is most suitable.

Retrieving a token

In order to execute transactions on eBay on behalf of a user, the application needs to have a token for that user. Up to this point in the tutorial, you've been using a token that was generated specifically for the application by the eBay token-generation tool. You don't use the token tool to generate tokens for users. eBay has an authentication and authorization process for token generation.

The first step in generating a token is to generate a secret ID and redirect the user to a URL that eBay provides. The application is responsible for adding a couple of query parameters to the URL before the user is redirected. The application needs to supply its runame and a unique secret ID that will be used to retrieve the token. eBay suggests that the secret ID be a Universal Unique Identifier (UUID). PHP doesn't have a built-in function for generating UUIDs, but there is a PECL extension for generating them (see [Resources](#)). The process of token generation is described in detail in the Java tutorial and in the eBay developer zone (see [Resources](#)).

After a user has registered with Livrivore and been redirected to eBay to agree to the terms and conditions, eBay generates an authentication token to be used by Livrivore for any API calls made on behalf of the user. At this point, let's assume the user has authenticated against eBay and a token has been generated. Livrivore needs to retrieve that token and store it for subsequent requests. [Listing 9](#) shows how to do it with `Services_Ebay`.

Listing 9. FetchToken example

```
<?php
require_once 'config.php';
require_once 'Services/Ebay.php';

list($userId, $secretId) = getUserInfoFromSomewhere();

$session = Services_Ebay::getSession($devId, $appId, $certId);
$session->setAuthenticationData($userId);
$ebay = new Services_Ebay($session);
$token = $ebay->FetchToken($secretId);
}
?>
```

A hypothetical method called `getUserInfoFromSomewhere()` returns the user's ID and the secret ID you used for the token generation. Most likely, this information was stored in a database when the user first registered. However, because eBay generates a token within 30 seconds of the user accepting the terms and conditions, it's possible to store the secret ID and user ID in a session and not persist the user until a token has been successfully generated.

The session-creation step is exactly the same as in the previous example; however, instead of calling the `setToken` method on the session, you call the `setAuthenticationData` method. The reason is obvious -- you don't have a token yet. Also be aware that the authentication user ID is the actual user and not the application. In other words, the call is made on behalf of the user to retrieve their token rather than being made on behalf of the application to retrieve a user's token.

Almost every API call is done on behalf of a user. I can't think of a call that can't be done with a user's credentials. For some calls, you may wish to use your application's credentials -- for example, Livrivore could display the ten most

expensive books currently selling on eBay on its front page. It's not doing that on behalf of a particular user, so it uses its authentication information. Almost certainly, any transaction that results in fees charged to the user will be done with their credentials.

Further, almost every API call is done with a token instead of a user ID and password combination. The exceptions are as follows:

- Getting a runame for your application (`GetRuName`)
- Getting the return URL for your application (`GetReturnURL`)
- Setting the return URL for your application (`SetReturnURL`)

The `FetchToken` API call is the only call that is made with just the user ID and no password. The reason is that the runame and return URL calls are made by the application, and it knows its own password; but the `FetchToken` call is for a user, and the application isn't allowed to know a user's password.

The last step is to invoke the `FetchToken` method on the `Services_Ebay` object with the secret ID. Doing so returns the token as a string, which you can then put in permanent storage.

Listing an item using a call

Now that you can retrieve user's tokens, you're able to conduct transactions on their behalf. The first kind of transaction you want to do for them is to let them list an item. (A note on terminology: The phrase "list an item" means the same as "add an item". It doesn't mean to search for or display an item.) [Listing 10](#) shows how to list an item with `Services_Ebay`.

Listing 10. AddItem example

```
<?php
require_once 'config.php';
require_once 'Services/Ebay.php';

$session = Services_Ebay::getSession($devId, $appId, $certId);
$session->setToken($token);
$ebay = new Services_Ebay($session);

$item = $ebay->loadModel("Item");
$item->Title = "Advanced PHP Programming";
$item->Description = "A practical guide to developing
    ." large scale web sites and applications with PHP5.";
$item->Quantity = 1;
$item->MinimumBid = 10.00;
$item->Location = 'us';
$item->Category = 377; // non-fiction
$item->PaymentSeeDescription = true;
if($ebay->AddItem($item)) {
```

```
    echo "Item added.\n";
    echo "You have been charged a listing fee of:"
      . " \${$item->Fees['ListingFee']}\n";
  }
?>
```

In the example where you retrieved a user's token with the `FetchToken` API call, the secret ID was a `string` and the token was returned as a `string`. Most API calls consume and return data that is more complex than primitive types like strings and numbers. `Services_Ebay` represents the types of entities that eBay uses as model classes.

Model classes serve two purposes: they provide an object-oriented interface to the XML that eBay deals with, and they often have helper methods that encapsulate some of the common behaviors performed on their models.

[Listing 10](#) shows the standard pattern for using models and calls. Even though a model is a standard PHP class -- in fact, the models are all subclasses of `Services_Ebay_Model` -- you don't instantiate it with the `new` operator. Instead, the `Services_Ebay` class serves as a model Factory. Calling the `loadModel` method on an instance of `Services_Ebay` with the name of your model creates an instance and initializes it appropriately. You should never instantiate any of the model classes directly.

`$ebay->loadModel("Item")` returns a blank model for you to populate. Even though the name of the method is `loadModel`, the method creates a new instance. Calling `loadModel` multiple times with the same parameter returns distinct instances.

The next step is to set the attributes for the listed item. Listing items on eBay is nontrivial. There are a lot of required fields and even more options. The attributes set here demonstrate the minimum necessary in order for the listing to be valid. A normal listing will have many more.

To set attributes, use the arrow operator and the name of the attribute you want to set. The name of the attribute is exactly the name of the element that would appear in the XML if you were using the test tool. To determine what attributes to set and what their names are, you need to either go to the API documentation or use the `Services_Ebay describeCall()` method discussed in [Error handling and debugging](#). In this case, I went to [AddItemInputArguments](#) and determined which arguments were required and the `ArgumentName` to use.

Services_Ebay and "magic methods"

I'm going to digress into the implementation details for a minute. Usually, you shouldn't need to know how `Services_Ebay` does things behind the scenes, but setting model properties can be tricky. The model classes use the PHP5 "magic methods" `__get` and

```
__set. When you say something like $item->Greeting = 'Hello, Nurse!', the Services_Ebay_Model class sticks an entry in its properties array with the key Greeting and the value Hello, Nurse!. When you make the API call, the properties array is turned into XML, and $item->Greeting = 'Hello, Nurse!' becomes <Greeting>Hello, Nurse!</Greeting>. This is sent to eBay along with all the other properties. eBay ignores any elements it doesn't understand and processes the ones it does.
```

```
This can lead to some subtle bugs. When I first created this example, I used StartPrice instead of MinimumBid because StartPrice is what it's called in the Java SOAP SDK -- I foolishly assumed the name would be the same. I set the value to 100 and made the call. When I checked the sandbox to see if it was correctly listed, everything was as expected, but the minimum bid was $1. I was perplexed because the API seemed to have been changed, so the monetary unit was the penny instead of the dollar. It turns out that the Item model sets some default values, including MinimumBid.
```

When you're finished setting the attributes on the model, you pass it in as a parameter to the API call. The call is executed, and any changes to the model are made based on the results from eBay. In this example, you can see that a successful listing results in the `Fees` attribute being populated with all the fees that eBay charges for listing (`ListingFee` being but one).

Listing an item using a model

You can look at the eBay API two ways. The first way sees the API as a bunch of functions that take and return data. This is a very procedure-centric view. The alternate view sees a bunch of entities that support different behaviors. This is an object-centric view. For example, in the procedure-centric view, procedures like `GetUser`, `GetFeedback`, and `GetSellerList` all take a user (as well as other things) as their input. Alternately, there is a `User`, and the `User` supports behavior that allows you to get its feedback, the items it is selling, and so on.

eBay has implemented its services the procedure-centric way, and everything I've talked about so far has used that approach. However, `Services_Ebay` also supports the other approach. [Listing 11](#) shows the previous example rewritten in the object-centric style.

Listing 11. AddItem in an object-oriented style

```
<?php
require_once 'config.php';
require_once 'Services/Ebay.php';

$session = Services_Ebay::getSession($devId, $appId, $certId);
$session->setToken($token);
$ebay = new Services_Ebay($session);
```

```

$item = $ebay->loadModel("Item");
$item->setSession($session);
$item->Title = "Advanced PHP Programming";
$item->Description = "A practical guide to developing large"
    ." scale web sites and applications with PHP5.";
$item->Quantity = 1;
$item->MinimumBid = 10.00;
$item->Location = 'us';
$item->Category = 377; // non-fiction
$item->PaymentSeeDescription = true;

if($item->Add()) {
    echo "Item added.\n";
    echo "You have been charged a listing fee of: "
        ."\${$item->Fees['ListingFee']}\n";
}
?>

```

The code contains exactly two differences. First, you need to explicitly pass the session to the item. Second, you invoke the `Add()` method on the `$item` instead of invoking `$ebay->AddItem($item)`.

From the standpoint of clarity and code length, there's not much difference. Personally, I think the object-centric style is more natural. [Listing 12](#) shows an example of a script that pulls back all the auctions for a user and cancels all auctions that don't have any bids.

Listing 12. Cancel open auctions example

```

<?php
// include standard header stuff
$end_time_from = gmdate("Y-m-d H:i:s", time()); // now
$end_time_to = gmdate("Y-m-d H:i:s", time()
    + (60 * 60 * 24 * 7)); // one week from now

$callParams = array( 'DetailLevel' => 32, // lots of detail
    'PageNumber' => 1,
    'ItemsPerPage' => 200,
    'EndTimeFrom' => $end_time_from,
    'EndTimeTo' => $end_time_to
);

$user = $ebay->loadModel("User");
$user->setSession($session);
$user->UserId = 'librivore';

$items = $user->GetSellerList($callParams);
echo "Retrieved ".count($items)." items\n";
foreach($items as $item) {
    if($item->BidCount == 0) {
        echo "Ending auction for item # {$item->ItemId}\n";
        $item->End(1); // 1 == "Lost or Broken". Oops!
    } else {
        echo "Not ending auction for item # "
            ."{ $item->ItemId}: { $item->BidCount} bid(s)\n";
    }
}
?>

```

There are two ways of determining what methods a model object supports -- either look at the API documentation (resources) or use the source.

Working with the raw XML

Sometimes there are benefits to using XML directly instead of abstracting it away into one or more model classes. For the cost of a bit of inconvenience, `Services_Ebay` can do that as well. [Listing 13](#) shows how to determine the number of items that are listed with the word "book" in the title and that are available as "buy it now" purchases.

Listing 13. GetSearchResults and parse XML example

```
<?php
//standard header
$params['Query'] = 'book';
$xml = $session->sendRequest('GetSearchResults',
    $params, Services_Ebay::AUTH_TYPE_TOKEN, false);
$xml = simplexml_load_string($xml);
$result = $xml->xpath('//BuyItNow[.= 1]');
echo "There are ".count($result)." books that you can buy now!";
?>
```

Instead of using the `GetSearchResults` call class, which would return an array of `Item`, you call the session's `sendRequest` method directly. Because you aren't using the call class, you're responsible for specifying the verb and creating an array of parameters. Remember, the request XML is always flat, so all you need for the parameters is an array with the input arguments as the keys and the argument values as the values.

The last two parameters to the `sendRequest` method are the authentication method and a `boolean` indicating whether the XML should be unmarshalled. As mentioned earlier, the authentication method is almost always `AUTH_TYPE_TOKEN`. If the unmarshalling parameter is `false`, `sendRequest` returns the raw XML; otherwise it returns an array containing the values extracted from the XML.

Because you have the XML, you can use the awesome power of XPath to slice and dice it in ways that would be difficult if you had the object graph of arrays and primitive types the session would normally return. The XPath expression `//BuyItNow[.= 1]` returns an array of all the nodes named `BuyItNow` with a value of 1. You use the `count` function to get the size of the array and display it.

SimpleXML XPath return values

Note for XPath users: You might wonder why I didn't make the

XPath expression `count(//BuyItNow[.= 1])` and dispense with the array-count function call. The reason is that the SimpleXML XPath API returns an array of `SimpleXMLElement` objects, and `count(//BuyItNow[.= 1])` returns a zero-element array rather than the integer value of the expression. There may be a way around it, but I don't know what it is.

Defining your own model

The model classes that `Services_Ebay` provides are sufficient for most purposes, but sometimes you'll need to modify or extend a model to fit your application. For example, `Librivore` is used for listing books on eBay. The book information that `Librivore` captures contains attributes like title, author, and ISBN. The `Item` model has attributes like item title and description. The problem is, your domain has an entity that is listable, like an `Item`, but is richer than the generic `Services_Ebay_Model_Item` class.

One way of dealing with this difference is to have a `Book` class that encapsulates the title, author, ISBN, and other book-specific attributes and use the `Item` model for the eBay-specific attributes. You would link the `Book` instances with the `Item` instances by using the eBay-generated `ItemId` as a key.

This isn't an optimal solution because functions that need to operate on eBay-specific attributes and book-specific attributes need to manage two different objects. A cleaner solution would be to make a `ListableBook` class that extends the `Item` class with the necessary book attributes.

In principle, it's simple to implement this solution. You need to create a class that subclasses the model class you want to extend, and you need to register it with the `Services_Ebay` class (see [Listing 14](#)).

Listing 14. ListableBook

```
<?php
    require_once 'Services/Ebay/Model/Item.php';

    class ListableBook extends Services_Ebay_Model_Item {
        public $Title; // upcased like other attributes
        public $Isbn;
        public $Author;
    }
?>

<?php
    require_once 'Services/Ebay.php';
    Services_Ebay::useModelClass('Item', 'ListableBook');
?>
```

You need to register your class with `Services_Ebay` because the model classes are never instantiated directly. The `Services_Ebay` class acts as a model factory for code both external and internal to the `Services_Ebay` framework.

I said this is simple in principle because there are a couple of things you need to be aware of. Registering the `ListableBook` class as an `Item` means that `Services_Ebay` treats every item as a `ListableBook`. For `Librivore`, this isn't a problem because the only items it lists are books. However, if `Librivore` sold books and Hummel figurines, you'd be in trouble.

The other issue you need to be aware of is the way model classes internally represent data. The `Services_Ebay_Model` superclass uses the `__get` and `__set` magic methods to store the properties of the object. If the `ListableBook` didn't have the `$Title`, `$Isbn`, and `$Author` instance variables, it would still be valid to have code like the following:

```
<?php
    Services_Ebay::useModelClass('Item', 'ListableBook');
    $session = Services_Ebay::getSession($devId, $appId, $certId);
    $listableBook = Services_Ebay::loadModel('Item');
    $listableBook->Title = "Advanced PHP Programming";
    $listableBook->setSession($session);
    $listableBook->Add();
?>
```

But because there isn't a `$Title` attribute, the `Services_Ebay_Model` `__set` method will be called; it will add a `Title` attribute to the parameter array that will be sent to eBay. If you're creating model classes that add attributes to existing models, take care to separate the data that is internal to your application from the data that gets sent to eBay.

Summary

At this point, you should have a good understanding of how to use the `Services_Ebay` framework to interact with the eBay Web services. You've seen how to do the following:

- Initialize `Services_Ebay`
- Authenticate through `Services_Ebay`
- Create model objects
- Execute API calls
- Work directly with the response XML

- Define and substitute your own model classes

The actual practice of interacting with eBay is rich and complex. This section has barely scratched the surface of the things you can do with eBay. Refer to the [Resources](#) section for more information.

The next section looks at handling errors and debugging applications that use `Services_Ebay`.

Section 5. Error handling and debugging

Here's where I come clean and admit that I haven't been doing a very good job of demonstrating good coding practices. I've been assuming that everything is going to go just fine and there won't be any problems with bad data, missing parameters, lost network connectivity, and all the other things that happen in the real world. In practice, things do go wrong, and you need to know how to find out what happened and how to either fix it or die gracefully.

Exceptions

`Services_Ebay` uses the standard PHP5 Exception extension. Every nonattribute method call can potentially throw an exception. Therefore, you should always invoke methods within a `try/catch` block (see [Listing 15](#)).

Listing 15. Using try/catch

```
<?php
$user = $ebay->loadModel("User");
$user->setSession($session);
$user->UserId = 'librivore';
try {
    $items = $user->GetSellerList($callParms);
    echo "Retrieved ".count($items)." items\n";
} catch (Exception $e) {
    echo "Unable to retrieve items: {$e->getMessage()}\n";
}
?>
```

`Services_Ebay` uses four different kinds of exceptions:

- `Services_Ebay_Exception`

- `Services_Ebay_Auth_Exception`
- `Services_Ebay_Transport_Exception`
- `Services_Ebay_API_Exception`

The first of the four is the superclass of the others. It's also used as a generic exception -- for example, if you pass too many parameters to a service call, you'll get this exception. The second and third are self-explanatory. The last one is used when eBay returns an error from your API call. `Services_Ebay` checks to see whether eBay has returned any errors of severity level 2 and, if so, throws a `Services_Ebay_API_Exception` with the messages from those errors.

Another thing you can do during the development phase is to use PHP's built-in error reporting functionality. Putting `error_reporting(E_ALL)` at the top of your scripts propagates any uncaught exceptions to standard out. This at least lets you see that there's a problem, rather than failing silently. The `E_ALL` setting also notifies you about any other potential problems like uninitialized variables. Personally, I find `error_reporting(E_ALL)` to be helpful during development. However, you should be aware that having it in production code is a security risk: it gives attackers information about the internals of your application that they can use to exploit it. When you move your application into production, redirect error reporting to a log file rather than standard out. Also, if you don't want to pepper your code with `error_reporting` calls, you can set the default level in your `php.ini` file.

Retrieving errors from the session

The exception-handling mechanism is helpful, but it doesn't mesh perfectly with eBay's error-reporting mechanism. Suppose you send the following XML through the test tool:

```
<foo/>
```

eBay has no idea what to do with this request, and it tells you so:

```
<?xml version="1.0" encoding="utf-8" ?>
<eBay>
  <EBayTime>2005-08-14 03:18:05</EBayTime>
  <Errors>
    <Error>
      <Code>2</Code>
      <ErrorClass>RequestError</ErrorClass>
      <SeverityCode>1</SeverityCode>
      <Severity>SeriousError</Severity>
      <Line>0</Line>
      <Column>0</Column>
      <ShortMessage><![CDATA[ Unsupported verb. ]]></ShortMessage>
```

```

    </Error>
  </Errors>
</eBay>

```

Whenever eBay returns a response with an `<Errors>` element, the `Services_Ebay_Session` picks it apart and makes the errors available -- but only if you ask. [Listing 16](#) calls `FetchToken` with an invalid secret ID.

Listing 16. Valid request with errors

```

<?php
    error_reporting(E_ALL);

    require_once 'config.php';
    require_once 'Services/Ebay.php';

    list($userId, $secretId) = getUserInfoFromSomewhere();

    $session = Services_Ebay::getSession($devId, $appId, $certId);
    $session->setAuthenticationData($userId);
    $ebay = new Services_Ebay($session);
    try {
        $token = $ebay->FetchToken($secretId);
    } catch (Exception $e) {
        echo $e->getMessage();
        $errors = $session->getErrors();
        print_r($errors);
    }

    function getUserInfoFromSomewhere() {
        return array("librivore", "bogus SID");
    }
?>

```

`$ebay->FetchToken()` throws an exception, which you catch. You echo the message in the exception and then retrieve the errors from the session. The session returns the errors in an array, even if there's only one. `print_r` pretty-prints the object to standard out. [Listing 17](#) shows the output from executing this script.

Listing 17. Response dumped with print_r

```

The secret Id you provided does not match with the one you provided
earlier.
Array
(
    [0] => Services_Ebay_Error Object
        (
            [code:private] => 16117
            [severityCode:private] => 1
            [severity:private] => SeriousError
            [shortMessage:private] => The secret Id is invalid.
            [longMessage:private] => The secret Id you provided does not
match with the one you provided earlier.
        )
)

```

The first line is the message from the exception. It's the same as the `<LongMessage>` from the XML. If there were multiple errors, the message would be the concatenation of all the `<LongMessage>`s separated with new lines.

The `Services_Ebay_Error` class is an object representation of the `<Error>` element and its children. You can see the correspondence between the dump of the object and the XML.

Incidentally, calling `getErrors()` has the side effect of clearing out the errors. If you call it twice in a row, the second call returns an empty array. This is usually desirable -- you don't want old errors hanging around after other calls. However, this behavior is counterintuitive. You don't usually expect getters to delete the data after returning it. The `getErrors()` method takes an optional boolean parameter. If you pass in `false`, it doesn't clear the errors.

Retrieving the request/response XML from the session

Sometimes, something goes wrong and neither the exceptions nor the errors from the session help. You need to see at a very low level exactly what data you're sending to eBay and what it's sending back. You can do this through the `getWire()` method on the session. `getWire()` returns the request XML that was sent to eBay as well as the response XML from eBay. It's disabled by default, so if you want to use it, you need to enable it by putting the session in debug mode (see [Listing 18](#)).

Listing 18. Putting the session in debug mode

```
<?php
// standard header
$session = Services_Ebay::getSession($devId, $appId, $certId);
$session->setAuthenticationData('librivore');
$session->setDebug(Services_Ebay_Session::DEBUG_STORE);
$ebay = new Services_Ebay($session);
try {
    $token = $ebay->FetchToken('bogus SID');
} catch (Exception $e) {
    echo $session->getWire();
}
?>
```

[Listing 19](#) shows the output.

Listing 19. Results from calling `getWire()`

```
Sending request:
```

```

<?xml version="1.0" encoding="UTF-8"?>
<request xmlns="urn:eBayAPISchema">
  <DetailLevel>0</DetailLevel>
  <ErrorLevel>1</ErrorLevel>
  <SiteId>0</SiteId>
  <Verb>FetchToken</Verb>
  <RequestUserId>librivore</RequestUserId>
  <SecretId>bogus SID</SecretId>
</request>

Received response:
<?xml version="1.0" encoding="utf-8" ?><eBay>
<EBayTime>2005-08-14 03:48:39</EBayTime>
<Errors>
<Error>
<Code>16117</Code>
<SeverityCode>1</SeverityCode>
<Severity>SeriousError</Severity>
<Line>0</Line>
<Column>0</Column>
<ErrorClass>RequestError</ErrorClass>
<ShortMessage><![CDATA[The secret Id is invalid.]]></ShortMessage>
<LongMessage>
<![CDATA[The secret Id you provided does not match with the one
you provided earlier.]]>
</LongMessage>
</Error>
</Errors>
</eBay>

```

`Services_Ebay_Session::DEBUG_STORE` tells the session to store the debug information for later retrieval. There is also a `Services_Ebay_Session::DEBUG_PRINT` value, but at the time of this writing the printing code hasn't been implemented.

Unlike `$session->getErrors()`, `$session->getWire()` doesn't clear the data when you call it. It does replace the old data with new data after every API call, though.

Services_Ebay API documentation

`Services_Ebay` has a couple of nice documentary pieces. The first has limited usefulness -- it lets you query the `Services_Ebay` class to see what API calls it supports: `php -r "require_once 'Services/Ebay.php'; print_r(Services_Ebay::getAvailableApiCalls());"`. It has limited usefulness because it's faster and easier (on my machine, at least) to type `ls /usr/local/lib/php/Services/Ebay/Call`, especially once tab-completion is taken into account.

The other feature is pretty cool, though: each call class is self-documenting. That means you can say the following:

```
$call = Services_Ebay::loadAPICall($callName);
```

```
$call->describeCall();
```

and a description of the call is printed to standard out. The following is what it looks like for `getUser`:

```
API Call : GetUser
Parameters (max. 2)
  1. UserId(no default value)
  2. ItemId(no default value)
API Documentation : http://developer.ebay.com/DevZone/docs/API_Doc/
Functions/GetUser/GetUserLogic.htm
```

If you have an extensible editor that lets you execute commands and then captures the output, this feature is particularly nice. You could create a macro that takes your selection and passes it in as input to the following command:

```
php -r "require_once('Services/Ebay.php');
Services_Ebay::loadAPICall($SELECTED_TEXT)->describeCall();" 
```

Bind it to a key sequence, and you have API documentation at your fingertips. Of course, substitute the `$SELECTED_TEXT` variable for whatever method your editor uses to expose the selected text.

Summary

`Services_Ebay` uses the PHP5 exception-handling system for internal exceptions as well as eBay API errors. At this point, you should know the following:

- The different exception classes `Services_Ebay` uses and what they mean
- How to retrieve any errors that eBay returns
- How to put the session in debug mode
- How to get the request and result XML from the session

There's nothing left now but to wrap-up.

Section 6. Conclusion

This tutorial discussed how to use PHP to develop applications using the eBay XML API. You should feel comfortable with the implementation details of invoking eBay Web services. You learned how to do the following:

- Write your own eBay XML API test tool.
- Register users and retrieve their authentication tokens.
- List items with `Services_Ebay`, either as an API call or as a behavior on the `Item` model.
- Execute eBay searches through the API.
- Capture the response XML and bypass the models in `Services_Ebay`.
- Retrieve and handle API errors.
- Debug calls made through `Services_Ebay`.
- View the internal documentation in each `Services_Ebay` call class.

This should give you a good start in developing your own eBay applications, but I've only scratched the surface. The eBay Developer Zone has thousands of pages of documentation, a good portion of which is about the eBay business model. I haven't touched much on this subject, but it's important to understand if you plan to create a commercial application.

I hope you've enjoyed this tutorial -- and I'm very interested in any feedback you might have.

Section 7. Appendix: Migrating to the next schema

I have some bad news. Everything I've talked about with respect to the XML schema is obsolete. That's right: eBay is deprecating the XML API I've looked at in such detail. The good news is that application developers have until June 1, 2006 to make the migration.

I discussed the deprecated API because it's currently used by most applications that use the eBay XML API, particularly `Services_Ebay`. Illustrating the new API wouldn't be helpful to you if you needed to use the `Services_Ebay` debugging facilities. The schema changes are important only if you're developing an application that deals with the XML directly. If you develop an application that uses `Services_Ebay`, you can safely bet that it will be modified to use the new schema.

The bad news is that `Services_Ebay` is tied closely to the XML representation. When you say `$user->UserId = 'librivore'`, `Services_Ebay` creates an element in the XML document called `<UserId>`. If you want to minimize the changes to your application, provide an interface layer between your application's business logic and `Services_Ebay`. That way, when `Services_Ebay` changes to match the new schema, you'll only need to change the interface layer and not your whole application. In other words, try to minimize the dependencies between your code and any unstable code you decide to use.

Here's how to write the `GetUser` XML in the new schema:

```
<GetUserRequest xmlns="urn:ebay:apis:eBLBaseComponents">
  <RequesterCredentials>
    <eBayAuthToken>ABC...123</eBayAuthToken>
  </RequesterCredentials>
  <UserID>sampleuser</UserID>
</GetUserRequest>
```

The primary difference is that the root element of the document is the verb prepended to `Request` instead of the value of the `<Verb>` element.

The response has been changed in a similar fashion:

```
<GetUserResponse xmlns="urn:ebay:apis:eBLBaseComponents">
  <Timestamp>2005-03-18T02:58:59.289Z</Timestamp>
  <Ack>Success</Ack>
  <CorrelationID>00000000-00000000-...-00000000-0000000000</
CorrelationID>
  <Version>397</Version>
  <Build>20050316222508</Build>
  <User>
    <AboutMePage>false</AboutMePage>
    <EIASToken>nY+sHZ2PjriAZeEqQ2d8jx9nY+seQ==</EIASToken>
    <Email>sampleuser@foobar.com</Email>
    <FeedbackScore>15</FeedbackScore>
    <FeedbackRatingStar>Yellow</FeedbackRatingStar>
    .
    .
  </User>
</GetUserResponse>
```

Resources

- [Develop apps with Web services and the eBay SDK, Part 1](#) and [Part 2](#): Before you start this tutorial, you may find it helpful to read two other eBay tutorials. The first one describes how to build a command-line eBay search engine in Java. The second one demonstrates how to build a book-distribution application with the eBay Java SDK and provides detailed information about the eBay authentication and authorization mechanism in a language-agnostic way.
- [PHP Manual](#): A good place to start if you're just learning PHP. For this tutorial, you might want to skim through the following sections if you need a refresher: [Classes and Objects](#), [SimpleXML](#), [CURL](#), and [utf8_encode](#).
- There are a lot of good books on PHP, but I particularly recommend [Beginning PHP 5 and MySQL: From Novice to Professional](#), [Beginning PHP 5 and MySQL E-Commerce: From Novice to Professional](#), and [Advanced PHP Programming](#). Additionally, Jack Herrington has written a good article on the [PHP scalability myth](#).
- [Services_Ebay](#) documentation and resources are available at its [PEAR repository](#) and through the [eBay Community Codebase](#).
- One of the benefits of the eBay XML API is that you can use XML techniques and frameworks. In particular, XPath is very good for extracting information. Chapter 9 of [XML in a Nutshell](#) gives a good overview of XPath, and it's [online](#).
- The best place to learn about writing eBay applications is at the [eBay Developer Zone](#). That's where you'll find documentation for the [XML API](#) that [Services_Ebay](#) uses as well as eBay's [new XML API](#). eBay also has [forums](#) where developers can ask questions and a [Knowledge Base](#) with information that's not always in the documentation. In addition, [whitepapers](#) provide some of the business strategy for eBay's Web services.
- In addition to the educational resources that eBay provides, the company also has tools that make development easier: [create a sandbox user](#), [create a single-user authentication token](#), and [use the XML API test tool](#).

About the author

Alex Garrett

After pursuing an undergraduate degree in classics, linguistics, computer science, psychology, and literature, Alex finally settled on a B.S. in philosophy from the University of Wisconsin. His professional career is as checkered as his academic one. He has been the e-commerce architect for a GE Capital company, a lecturer at a University of Wisconsin school, a systems administrator, an acquisitions

editor for a small technical publishing company, and a code toad, among other things. Currently, he's a senior consultant with a Madison, WI-based firm and a proud new father.