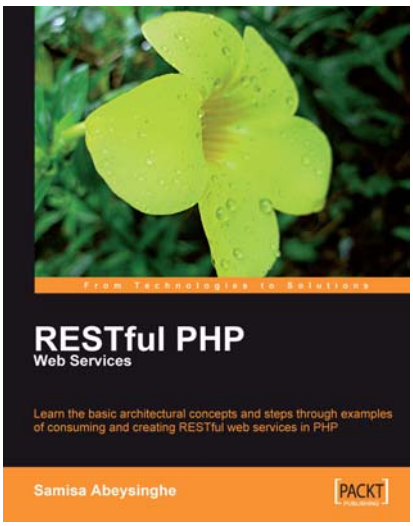
RESTful PHP Web Services

Samisa Abeysinghe



Chapter No. 5
"Resource-Oriented Clients"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.5 "Resource-Oriented Clients"

A synopsis of the book's content

Information on where to buy this book

About the Author

Samisa Abeysinghe is Director, Engineering at WSO2. Samisa pioneered the Apache Axis2/C effort and architected the core of the Apache Axis2/C Web services engine. He continues to be an active contributor in the Apache Axis2/C project. His involvement in open-source projects began in 2004 when he started working with the Apache Axis C/C++ project.

Prior to his current role, Samisa played the project lead role for the WSO2 Web Services Framework for PHP project, which provides comprehensive support for building both SOAP and REST services and clients.

"No man is an island"—John Donne

As human beings, we do not thrive when isolated from others. This book was no exception. Many people contributed to the successful completion of this book, and I would like to acknowledge all those who contributed.

First, I must thank Douglas Paterson. Douglas, Senior Acquisition Editor of Packt Publishing Ltd., is the one who initially proposed to me that I write this book. And thanks to him, this book was born.

Next, my gratitude goes to Sanjiva Weerawarana, Founder, Chairman, and CEO ofWSO2, Inc. When I first consulted Sanjiva on his thoughts on whether I should be writing this book, he encouraged me and even offered to help.

Speaking about encouragement, I must thank my mother, who checked, on a weekly basis, if I was continuing with my work on the book and the progress that I was making on that front.

The staff at Packt Publishing Ltd. helped a great deal to make this book a reality. I would like thank Rajashree Hamine the project coordinator, Swapna Verlekar the development editor, and Siddharth Mangarole the technical editor. I would also like to thank all others from Packt Publishing Ltd. who contributed to this book in many ways.

I would also like to thank some of my WSO2 colleagues, who worked with me closely on the scripting projects, specially WSO2 WSF/PHP. I would like to mention Nandika, Dimuthu, Chinthana, and Buddhika. Though they did not work on this book directly, they helped me a lot to understand PHP while working on WSF/PHP.

RESTful PHP Web Services

This book discusses the use of PHP to implement web applications based on REST architectural principles. Web services are a popular breed of web application technologies in today's programmable Web, and REST is the most popular style used in there. This book uses real-world examples as well as step-by-step guidelines to explain how to design REST-style services and clients from the ground up and how to use PHP programming constructs and frameworks to implement those services and clients.

What This Book Covers

Chapter 1 introduces the concepts related to the programmable Web, shows how HTTP and web services are related to each other, introduces the principles behind REST, explains how HTTP verbs are used in REST applications, explains the need for RESTFul web services while building PHP web applications, and introduces some frameworks and tools that can be used to work with REST in PHP.

Chapter 2 takes a first look at REST with PHP. While providing and consuming REST-style web services, the primary pre-requisites are an HTTP server or an HTTP client library and an XML parser library. In this chapter, we will see how to use the PHP CURL API to consume web services using various HTTP verbs such as `HTTP GET`, `POST`, `PUT`, and `DELETE`. The DOM API and SimpleXML API for building XML object structures and parsing XML streams are also discussed. We will discuss in detail how to build XML request payloads and also how to parse XML response payloads. The final section of this chapter demonstrates how to use the HTTP client features and XML parser features to invoke the Flickr REST API.

Chapter 3 looks into some real-world applications and discusses how to combine multiple service interfaces to build value-added custom applications. In this chapter, we will see how to use RSS or ATOM feeds, Yahoo search API, and Yahoo maps API. With the know-how you gain in this chapter and the previous chapters, you could build very powerful value-added applications like mashups using publicly available REST-style services.

Chapter 4 covers the steps that you would have to follow in designing and implementing a resource-oriented service in detail. Identifying resources and business operations for a given problem statement, designing the URI patterns, selecting the correct HTTP verbs, mapping URI and HTTP verbs to business operations are covered using the library example. Implementing the services and business operations using PHP is explained in detail, step by step.

Chapter 5 covers the steps that you would have to follow in designing and implementing resource-oriented clients in detail. The design of the clients is governed by the design of the service. And the client programmer needs to understand the semantics of the service, which is usually communicated through service API documentation. In the examples of this chapter, we will use the library service API designed in Chapter 4 to explain how we could use an existing API while designing PHP applications.

Chapter 6 uses the REST classes provided with the Zend Framework to implement the sample library system. The design of the service and client are covered, along with the MVC concepts supported by the Zend Framework. We will discuss how resources map to the model in MVC, and how HTTP verbs when combined with resource URIs map to the controller in MVC. We will explore how to combine `Zend_Rest_Server` with `Zend_Controller` to implement the business operations of the service and how to use `Zend_Rest_Client` class to consume the services.

Chapter 7 looks into the use of tools to trace and look into the messages to figure out possible problems with request and response pairs passed between clients and services. That helps with debugging and troubleshooting of services and clients. We will also look into how we could look at the XML documents to figure out possible problems in building XML in this chapter, and discuss how we can locate problems in parsing incoming XML messages.

Appendix A introduces the WSO2 Web Services Framework for PHP (WSO2 WSF/PHP) and discusses how to use the WSF/PHP service API to implement the sample Library system as a REST service and implement a REST client to consume it. We will also look into using the SOAP features provided in the frameworks to implement a SOAP client to consume the same service using SOAP-style messages. This chapter also discusses the differences between REST and SOAP message styles, in brief.

Appendix B introduces a PHP class named `RESTClient` that can be used to consume REST-style services. This class supports all key HTTP verbs, `GET`, `POST`, `PUT`, and `DELETE`. The advantage of using such a class is that it minimizes the complexity of your client code. At the same time, you can re-use this class for all your REST-style client implementations. This PHP class is sufficient for most simple REST-style client programs, and requires no third-party libraries. However, if you want to implement services and also want advanced clients, it is advised to use a more established framework such as Zend Framework or WSO2 WSF/PHP introduced in Chapter 6 and Appendix A of this book.

5
Resource-Oriented Clients

Resource-Oriented clients are client programs that consume services designed in accordance with the REST architectural principles. As explained in Chapter 1, the key REST principles include:

- The concept of resource (for example, a document is a resource)
- Every resource given a unique ID (for example, document URL)
- Resources can be related (for example, One document linking to another)
- Use of standard (HTTP, HTML, XML)
- Resources can have multiple forms (for example, status of a document, updated, validated, deleted)
- Communicate in a stateless fashion using HTTP (for example, subsequent requests not related to each other)

In the previous chapter, we studied in detail, and from ground-up, how to design and implement services to comply with REST architectural principles. In this chapter, we will study how we can implement clients to consume those services. We will use the same real-world example that we used in the last chapter, the simplified library system, to learn from scratch how to design clients with REST principles in mind.

Designing Clients

In the last chapter, while designing the library service, the ultimate outcome was the mapping of business operations to URIs and HTTP verbs. The client design is governed by this mapping.

Prior to service design, the problem statement was analysed. For consuming the service and invoking the business operations of the service using clients, there needs to be some understanding of how the service intends to solve the problem. In other words, the service, by design, has already solved the problem. However, the semantics of the solution provided by the service needs to be understood by the developers implementing the clients. The semantics of the service is usually documented in terms of business operations and the relationships between those operations. And sometimes, the semantics are obvious. As an example, in the library system, a member returning a book must have already borrowed that book. The `borrow book` operation precedes the `return book` operation. Client design must take these semantics into account.

Resource Design

Following is the URI and HTTP verb mapping for business operations of the library system that we came up with in the last chapter.

URI	HTTP Method	Collection	Operation	Business Operation
`/book`	`GET`	books	retrieve	`Get books`
`/book`	`POST`	books	create	`Add book(s)`
`/book/{book_id}`	`GET`	books	retrieve	`Get book data`
`/member`	`GET`	members	retrieve	`Get members`
`/member`	`POST`	members	create	`Add member(s)`
`/member/{member_id}`	`GET`	members	retrieve	`Get member data`
`/member/{member_id}/books`	`GET`	members	retrieve	`Get member borrowings`
`/member/{member_id}/books/{book_id}`	`POST`	members	create	`Borrow book`
`/member/{member_id}/books/{book_id}`	`DELETE`	members	delete	`Return book`

When it comes to client design, the resource design is given, and is an input to the client design. The resource design was covered in the last chapter where we designed the service design. When it comes to implementing clients, we have to adhere to the design given to us by the service designer. In this example, we designed the API given in the above table, so we are already familiar with the API. Sometimes, you may have to use an API designed by someone else, hence you would have to ensure that you have access to information such as:

- Resource URI formats
- HTTP methods involved with each resource URI
- The resource collection that is associated with the URI
- The nature of the operation to be executed combining the URI and the HTTP verb
- The business operation that maps the resource operation to the real world context

Looking into the above resource design table, we can identify two resources, book and member. And we could understand some of the semantics associated with the business operations of the resources.

- Create, retrieve books
- Create, retrieve members
- Borrow book, list borrowed books and return book
- Book ID and member ID could be used to invoke operations specific to a particular book or member instance

System Implementation

In this section, we will use the techniques that we discussed in the previous chapters on client programming to consume the library service we implemented in the last chapter. These techniques include:

- Building requests using XML
- Sending requests with correct HTTP verbs using an HTTP client library like CURL
- Receiving XML responses and processing the received responses to extract information that we require from the response

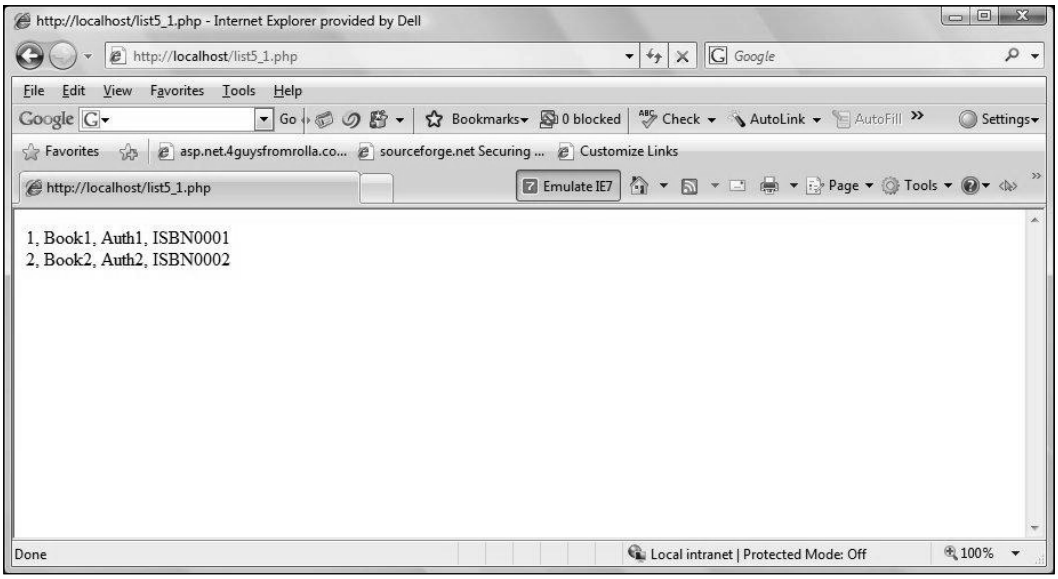
Retrieving Resource Information

Here is the PHP source code to retrieve book information.

```
<?php
$url = 'http://localhost/rest/04/library/book.php';
$client = curl_init($url);
curl_setopt($client, CURLOPT_RETURNTRANSFER, 1);
$response = curl_exec($client);
curl_close($client);
```

```
$xml = simplexml_load_string($response);
foreach ($xml->book as $book) {
   echo "$book->id, $book->name, $book->author, $book->isbn <br/>\n";
}
?>
```

The output generated is shown below.



As per the service design, all that is required is to send a GET request to the URL of the book resource. And as per the service semantics, we are expecting the response to be something similar to:

```
<books>
   <book>
      <id>1</id>
      <name>Book1</name>
      <author>Auth1</author>
      <isbn>ISBN0001</isbn>
   </book>
   <book>
      <id>2</id>
      <name>Book2</name>
      <author>Auth2</author>
      <isbn>ISBN0002</isbn>
   </book>
</books>
```

So in the client, we convert the response to an XML tree.

```
$xml = simplexml_load_string($response);
```

And generate the output that we desire from the client. In this case we print all the books.

```
foreach ($xml->book as $book) {
   echo "$book->id, $book->name, $book->author, $book->isbn <br/>\n";
}
```

The output is:

```
1, Book1, Auth1, ISBN0001
2, Book2, Auth2, ISBN0002
```

Similarly, we could retrieve all the members with the following PHP script.

```
<?php
$url = 'http://localhost/rest/04/library/member.php';
$client = curl_init($url);
curl_setopt($client, CURLOPT_RETURNTRANSFER, 1);
$response = curl_exec($client);
curl_close($client);
$xml = simplexml_load_string($response);
foreach ($xml->member as $member) {
   echo "$member->id, $member->first_name, $member->last_name <br/>\
n";
}
?>
```

Next, retrieving books borrowed by a member.

```
<?php
$url = 'http://localhost/rest/04/library/member.php/1/books';
$client = curl_init($url);
curl_setopt($client, CURLOPT_RETURNTRANSFER, 1);
$response = curl_exec($client);
curl_close($client);
$xml = simplexml_load_string($response);
foreach ($xml->book as $book) {
   echo "$book->id, $book->name, $book->author, $book->isbn <br/>\n";
}
?>
```

Here we are retrieving the books borrowed by member with ID 1. Only the URL differs, the rest of the logic is the same.

Creating Resources

Books, members, and borrowings could be created using `POST` operations, as per the service design. The following PHP script creates a new book.

```
<?php
$url = 'http://localhost/rest/04/library/book.php';
$data = <<<XML
<books>
    <book><name>Book3</name><author>Auth3</author><isbn>ISBN0003</
isbn></book>
    <book><name>Book4</name><author>Auth4</author><isbn>ISBN0004</
isbn></book>
</books>
XML;
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $url);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_POST, true);
curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
$response = curl_exec($ch);
curl_close($ch);
echo $response;
?>
```

When data is sent with `POST` verb to the URI of the book resource, the posted data would be used to create resource instances. Note that, in order to figure out the format of the XML message to be used, you have to look into the service operation documentation. This is where the knowledge on service semantics comes into play.

Next is the PHP script to create members.

```
<?php
$url = 'http://localhost/rest/04/library/member.php';
$data = <<<XML
<members><member><first_name>Sam</first_name><last_name>Noel</last_
name></member></members>
XML;
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $url);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_POST, true);
```

```
curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
$response = curl_exec($ch);
curl_close($ch);
echo $response;
?>
```

This script is very similar to the script that creates books. Only differences are the endpoint address and the XML payload used. The endpoint address refers to the location where the service is located. In the above script the endpoint address of the service is:

```
$url = 'http://localhost/rest/04/library/member.php';
```

Next, borrowing a book can be done by posting to the member URI with the ID of the member borrowing the book, and the ID of the book being borrowed.

```
<?php
$url = 'http://localhost/rest/04/library/member.php/1/books/2';
$data = <<<XML
XML;
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $url);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_POST, true);
curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
$response = curl_exec($ch);
curl_close($ch);
echo $response;
?>
```

Note that, in the above sample, we are not posting any data to the URI. Hence the XML payload is empty:

```
$data = <<<XML
XML;
```

As per the REST architectural principles, we just send a `POST` request with all resource information on the URI itself. In this example, the member with ID 1 is borrowing the book with ID 2.

```
$url = 'http://localhost/rest/04/library/member.php/1/books/2';
```

One of the things to be noted in the client scripts is that we have used hard coded URLs and parameter values. When you are using these scripts with an application that uses a Web-based user interface, those hard coded values need to be parameterized.

And we send a `POST` request to this URL:

```
curl_setopt($ch, CURLOPT_URL, $url);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_POST, true);
curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
```

Note that, even though the XML payload that we are sending to the service is empty, we still have to set the `CURLOPT_POSTFIELDS` option for CURL. This is because we have set `CURLOPT_POST` to `true` and the CRUL library mandates setting `POST` field option even when it is empty.

This script would cause a book borrowing to be created on the server side. As we saw in the last chapter, when the `member.php` script receives a request with the from `/{member_id}/books/{book_id}` with HTTP verb `POST`, it maps the request to `borrow book` business operation. So, the URL

```
$url = 'http://localhost/rest/04/library/member.php/1/books/2';
```

means that member with ID 1 is borrowing the book with ID 2.

Deleting Resources

We can use the `HTTP DELETE` operation to return the book.

```
<?php
$url = 'http://localhost/rest/04/library/member.php/1/books/2';
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $url);
curl_setopt($ch, CURLOPT_CUSTOMREQUEST, "DELETE");
curl_exec($ch);
curl_close($ch);
?>
```

In this case, we are sending a `DELETE` request with member ID and book ID in place. The above script indicates that member with ID 1 is returning the book with ID 2.

Putting it All Together

Writing the client scripts is trivial, all you have to do is:

- Identify the endpoint URI
- Find out the XML message format to be sent to service, if any
- Identify the expected HTTP verb to be used
- Send request
- Process response

Steps 1 to 3 would be found in service API documentation. In our example library system, we used a table to document our service API:

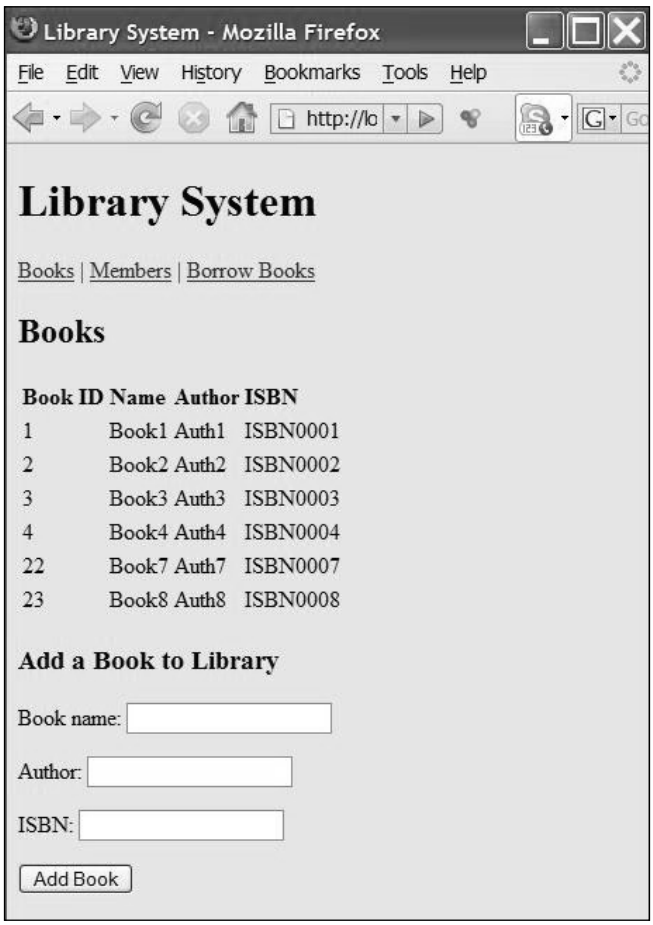
URI	HTTP Method	Collection	Operation	Business Operation
`/book`	`GET`	books	retrieve	`Get books`
`/book`	`POST`	books	create	`Add book(s)`

As we have seen in the previous chapters, all popular publicly available Web services have documentation available and those documents contain all these information.

Steps 4 and 5 consist of the use of HTTP client libraries and XML processing that we discussed in Chapter 2.

Implementing a Form-based Application

So far, we looked into the elements of PHP source code that would let us access the various operations of the library system. Now, let's see how we could put them all together.



The above picture shows an application built on top of the service interface provided by the library system. The main menu of this application is right below the main page title. You can view and add books, view and add members as well as borrow books and list the books borrowed by members. This application consists of one HTML file and three PHP scripts. The HTML file forms the main layout of the application.

Here is the index HTML file.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
   <meta http-equiv="content-type" content="text/html; charset=UTF-8">
   <title>Library System</title>
   <script language="javascript">
   <!--
   function fillContent(resource)
   {
       xmlhttp = new XMLHttpRequest();
       xmlhttp.open("GET", resource, true);
       xmlhttp.onreadystatechange=function() {
           if (xmlhttp.readyState==4) {
               document.getElementById('content').innerHTML = xmlhttp.
responseText;
           }
       }
       xmlhttp.send(null);
   }
   -->
   </script>
</head>
<body onload="fillContent('books.php')">
    <h1>Library System</h1>
   <div id="menu" align="left"><a href="#" onclick="javascript:
fillContent('books.php')">Books</a>
   | <a href="#" onclick="javascript:fillContent('members.
php')">Members</a>
   | <a href="#" onclick="javascript:fillContent('borrowings.
php')">Borrow Books</a>
   </div>
   <div id="content" align="left">
    </div>
</body>
</html>
```

This HTML code divides the browser window into two `div` areas, with the Ids `menu` and `content`.

The `div` section with the ID `menu`, displays the menu.

```
   <div id="menu" align="left"><a href="#" onclick="javascript:
fillContent('books.php')">Books</a>
   | <a href="#" onclick="javascript:fillContent('members.
php')">Members</a>
   | <a href="#" onclick="javascript:fillContent('borrowings.
php')">Borrow Books</a>
   </div>
```

Note that each menu item is linked to a JavaScript `onclick` action. The JavaScript action calls the JavaScript function `fillContent()` with the name of the PHP script to be loaded to the content `div`.

The `fillContent()` JavaScript function creates an `XMLHttpRequest` object instance and send a `GET` request to the PHP resource to load HTML content to the content `div`.

```
        xmlhttp = new XMLHttpRequest();
        xmlhttp.open("GET", resource, true);
```

Then once the content is received, when the `XMLHttpRequest` object's state changes to ready, the response text received is set as the inner HTML of the content div.

```
        xmlhttp.onreadystatechange=function() {
            if (xmlhttp.readyState==4) {
                document.getElementById('content').innerHTML = xmlhttp.
responseText;
            }
        }
```

Let us look at the PHP script that handles books.

We need to list the books and display them in a table. The following code does that.

```
   <table>
   <tr>
     <th> Book ID </th>
     <th> Name </th>
     <th> Author </th>
     <th> ISBN </th>
   </tr>
<!-- List Books -->
<?php
$url = 'http://localhost/rest/04/library/book.php';
$client = curl_init($url);
curl_setopt($client, CURLOPT_RETURNTRANSFER, 1);
$response = curl_exec($client);
```

```
curl_close($client);

$xml = simplexml_load_string($response);

foreach ($xml->book as $book) {
    echo "<tr> <td> $book->id </td> <td> $book->name </td> <td> $book-
>author </td> <td>$book->isbn </td></tr>";
}
?>
   </table>
```

We also need to have a form to facilitate the addition of a new book. Here is the source code for the form.

```
    <h2> Add a Book to Library </h2>

    <form>
        <form action="books.php" method="POST">
          <p>Book name: <input type="text" name="name" /></p>
          <p>Author: <input type="text" name="author" /></p>
          <p>ISBN: <input type="text" name="isbn" /></p>
          <p><input type="submit" name="submit" value="Add Book" /></
p>
    </form>
```

Now once the **Submit** button is clicked by the user, we post that data to the same PHP script, so the `create book` operation needs to be handled by the same PHP script. Now that needs to be done before we list the books, as we would like the new book too to appear in the list of books. Following is the PHP source code to do this.

```
<?php
if (isset ($_GET['name'])) {
    $url = 'http://localhost/rest/04/library/book.php';
    $data = "<books><book><name>" . $_GET['name'] . "</name><author>"
. $_GET['author'] .
    "</author><isbn>" . $_GET['isbn'] . "</isbn></book></books>";
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_URL, $url);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    curl_setopt($ch, CURLOPT_POST, true);
    curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
    $response = curl_exec($ch);
    curl_close($ch);
}
?>
```

This code looks if the `name` parameter is set,

```
if (isset ($_GET['name'])) {
```

And if it is set, we know that the user submits a request for a new book creation. So we pick the name, author, and ISBN from the parameters and from the data string to be posted for the `create book` operation.

```
    $data = "<books><book><name>" . $_GET['name'] . "</name><author>"
. $_GET['author'] .
    "</author><isbn>" . $_GET['isbn'] . "</isbn></book></books>";
```

Following is the full source code for this `books.php` script.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <meta http-equiv="content-type" content="text/html; charset=UTF-8">
</head>
<body>

<!-- Create book -->
<?php
if (isset ($_GET['name'])) {

    $url = 'http://localhost/rest/04/library/book.php';

    $data = "<books><book><name>" . $_GET['name'] . "</name><author>"
. $_GET['author'] .
    "</author><isbn>" . $_GET['isbn'] . "</isbn></book></books>";

    $ch = curl_init();

    curl_setopt($ch, CURLOPT_URL, $url);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    curl_setopt($ch, CURLOPT_POST, true);
    curl_setopt($ch, CURLOPT_POSTFIELDS, $data);

    $response = curl_exec($ch);

    curl_close($ch);
}
?>

   <h2>Books</h2>
   <table>
   <tr>
     <th> Book ID </th>
     <th> Name </th>
     <th> Author </th>
```

```
      <th> ISBN </th>
    </tr>
<!-- List Books -->
<?php
$url = 'http://localhost/rest/04/library/book.php';
$client = curl_init($url);
curl_setopt($client, CURLOPT_RETURNTRANSFER, 1);
$response = curl_exec($client);
curl_close($client);
$xml = simplexml_load_string($response);
foreach ($xml->book as $book) {
    echo "<tr> <td> " . htmlspecialchars($book->id) . "</td> ".
       " <td> " . htmlspecialchars($book->name) . "</td> " .
       " <td> " . htmlspecialchars($book->author) . " </td> ".
       " <td> " . htmlspecialchars($book->isbn) . " </td></tr>";
}
?>
    </table>
    <h3> Add a Book to Library </h3>
    <form>
        <form action="books.php" method="POST">
            <p>Book name: <input type="text" name="name" /></p>
            <p>Author: <input type="text" name="author" /></p>
            <p>ISBN: <input type="text" name="isbn" /></p>
            <p><input type="submit" name="submit" value="Add Book"
/></p>
    </form>
</body>
</html>
```

The `members.php` script does more or less the same job. It lists members and lets you add a new member.

Here is the full source code:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <meta http-equiv="content-type" content="text/html; charset=UTF-8">
</head>
<body>
```

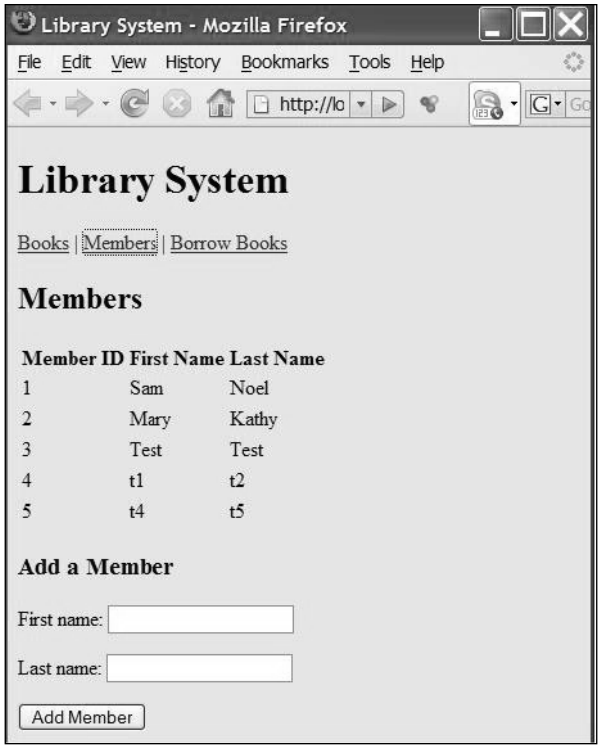
```
<!-- Add member -->
<?php
if (isset ($_GET['fname'])) {
    $url = 'http://localhost/rest/04/library/member.php';
    $data = "<members><member><first_name>" . $_GET['fname'] . "</
first_name><last_name>" .
          $_GET['lname'] . "</last_name></member></members>";
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_URL, $url);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    curl_setopt($ch, CURLOPT_POST, true);
    curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
    $response = curl_exec($ch);
    curl_close($ch);
}
?>
<!-- List members -->
   <h2>Members</h2>
   <table>
   <tr>
     <th> Member ID </th>
     <th> First Name </th>
     <th> Last Name </th>
   </tr>
<?php
$url = 'http://localhost/rest/04/library/member.php';
$client = curl_init($url);
curl_setopt($client, CURLOPT_RETURNTRANSFER, 1);
$response = curl_exec($client);
curl_close($client);
$xml = simplexml_load_string($response);
foreach ($xml->member as $member) {
    echo "<tr> <td> ". htmlspecialchars($member->id) . " </td> " .
         " <td> " . htmlspecialchars($member->first_name) . " </td> "
.
         " <td> " . htmlspecialchars($member->last_name) . " </td> </
tr>";
}
?>
    </table>
   <!-- Display the form -->
```

```
    <h3> Add a Member </h3>
    <form>
        <form action="members.php" method="POST">
            <p>First name: <input type="text" name="fname" /></p>
            <p>Last name: <input type="text" name="lname" /></p>
            <p><input type="submit" name="submit" value="Add Member"
/></p>
    </form>
</body>
</html>
```

The three main sections of this code are clearly marked with comments and are identical to those that we saw in the `books.php` script.

- `Add member`
- `List members`
- `Display add member form`

Here is what we would see on the display.



Finally, we have the `borrowings.php script`. This script displays the borrowings done by a member and also has a form that lets us borrow or return a book, providing the book ID and the member ID.

This script is slightly different from the books and members script of this client application. Listing the borrowings needs two resources to be accessed.

```
<?php
$url = 'http://localhost/rest/04/library/member.php';
$client = curl_init($url);
curl_setopt($client, CURLOPT_RETURNTRANSFER, 1);
$response = curl_exec($client);
curl_close($client);
$xml = simplexml_load_string($response);
foreach ($xml->member as $member) {
    echo "<tr> <td> $member->id </td> <td> $member->first_name </td>
<td> $member->last_name </td>";
   $url = "http://localhost/rest/04/library/member.php/$member->id/
books";
   $client = curl_init($url);
   curl_setopt($client, CURLOPT_RETURNTRANSFER, 1);
   $response = curl_exec($client);
   curl_close($client);
   $xml = simplexml_load_string($response);
   foreach ($xml->book as $book) {
      echo "<td> $book->id , $book->name</td>";
   }
   echo "</tr>";
}
?>
```

There are two `foreach` loops in this section of code. The outer loop accesses the list of members and lists them, while the inner loop access the borrowed books for that member and lists them.

In a functional design approach, access of the members and the books borrowed by a given member would have been broken into two separate functions. However, while designing a service API the information contained in a response resulting from an operation invocation would contain all applicable information which is related to that operation. This will make sure that the number of requests from client to the service would be minimized to get a business operation completed. This is considered good practice because less requests from client to service means less use of the network and that makes the application become faster.

The form in this script also is different. It has two **Submit** buttons, one for book borrowing and the other for returning.

```
<form>
    <form action="members.php" method="POST">
      <p>Member ID: <input type="text" name="m_id" /></p>
      <p>Book ID: <input type="text" name="b_id" /></p>
      <p><input type="submit" name="borrow" value="Borrow Book" />
      <input type="submit" name="return" value="Return Book" /></p>
</form>
```

Since the form can be submitted using either button, we have to take that into account while processing the submitted data.

We have to first check if it is the **borrow** button or the **return** button that was clicked.

```
if (isset ($_GET['borrow']) || isset ($_GET['return'])) {
```

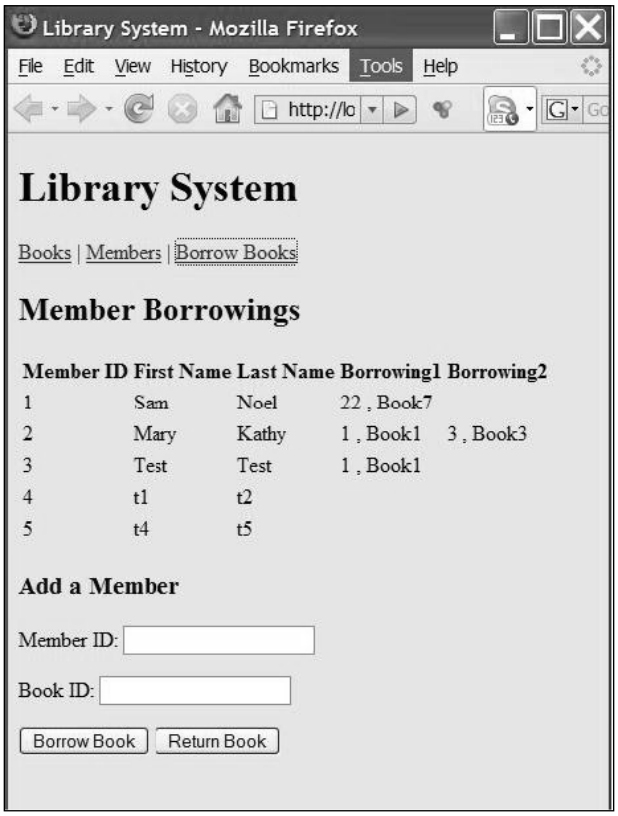
And based on that, we have to either perform a `POST` operation for borrowing or a `DELETE` operation for returning the resource. The resource URL should contain the book ID and the member ID.

```
$url = "http://localhost/rest/04/library/member.php/" . $_GET['m_id']
         "/books/" . $_GET['b_id'];
```

And we pick the correct HTTP verb to be used.

```
    if (isset ($_GET['borrow'])) {
       curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
       curl_setopt($ch, CURLOPT_POST, true);
       $data = "";
       curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
    }  else if (isset ($_GET['return'])) {
       curl_setopt($ch, CURLOPT_CUSTOMREQUEST, "DELETE");
    }
```

Here is the output.



And the complete source code for the borrowings PHP script is given below.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <meta http-equiv="content-type" content="text/html; charset=UTF-8">
</head>
<body>

<!-- Handle borrow or return book operations -->
<?php
if (isset ($_GET['borrow']) || isset ($_GET['return'])) {

    $url = "http://localhost/rest/04/library/member.php/" . $_GET['m_
id'] .
         "/books/" . $_GET['b_id'];
```

```
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_URL, $url);
    if (isset ($_GET['borrow'])) {
       curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
       curl_setopt($ch, CURLOPT_POST, true);
        $data = "";
       curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
    } else if (isset ($_GET['return'])) {
        curl_setopt($ch, CURLOPT_CUSTOMREQUEST, "DELETE");
    }
     curl_exec($ch);
     curl_close($ch);
}
?>
<!-- List book borrowings by members -->
    <h2>Member Borrowings</h2>
    <table>
    <tr>
      <th> Member ID </th>
      <th> First Name </th>
      <th> Last Name </th>
      <th> Borrowing1 </th>
      <th> Borrowing2 </th>
    </tr>
<?php
$url = 'http://localhost/rest/04/library/member.php';
$client = curl_init($url);
curl_setopt($client, CURLOPT_RETURNTRANSFER, 1);
$response = curl_exec($client);
curl_close($client);
$xml = simplexml_load_string($response);
foreach ($xml->member as $member) {
    echo "<tr> <td> " . htmlspecialchars($member->id) . " </td> ".
         "<td> " . htmlspecialchars($member->first_name) . " </td> " .
       "<td> " . htmlspecialchars($member->last_name) . " </td>";
   $url = "http://localhost/rest/04/library/member.php/$member->id/
books";
   $client = curl_init($url);
   curl_setopt($client, CURLOPT_RETURNTRANSFER, 1);
   $response = curl_exec($client);
   curl_close($client);
```

```
    $xml = simplexml_load_string($response);
    foreach ($xml->book as $book) {
       echo "<td> $book->id , $book->name</td>";
    }
    echo "</tr>";
}
?>
    </table>
    <!-- Display the form to borrow or return books -->
    <h3> Add a Member </h3>
    <form>
        <form action="members.php" method="POST">
            <p>Member ID: <input type="text" name="m_id" /></p>
            <p>Book ID: <input type="text" name="b_id" /></p>
            <p><input type="submit" name="borrow" value="Borrow Book"
/>
            <input type="submit" name="return" value="Return Book"
/></p>
    </form>
</body>
</html>
```

Summary

This chapter covered the steps that you would have to follow in designing and implementing resource-oriented clients in detail. The design of the clients is governed by the design of the service. And the client programmer needs to understand the semantics of the service, which is usually communicated through service API documentation. In the examples of this chapter, we used the library service API that we designed in the last chapter to explain how we could use an existing API while designing PHP applications.

In the next chapter, we will look into Zend framework's REST API.

Where to buy this book

You can buy RESTful PHP Web Services from the Packt Publishing website:
`http://www.packtpub.com/restful-php-web-services/book`

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com