PrestaShop 1.6 Web Service Tutorial

- Creado por Xavier Borderie, modificado por última vez en ene 24, 2014
- http://doc.prestashop.com/display/PS16/Web+service+tutorial

Web service tutorial

This tutorial explores how to create a small application to perform these four basic operations for customers.

Prerequisites

- PrestaShop 1.6 installed on a server with mod_rewrite enabled (Apache only).
- A local xAMP server with PHP5 enabled.

About CRUD & REST

The PrestaShop web service uses the REST architecture in order to be available on as many platforms as possible, since the HTTP protocol and XML files are understood by most platforms, if not all.
CRUD is an acronym that stands for "Create, Read, Update, and Delete". These are the four basic operations for managing data in an application.
REST defines roughly a style of software architecture, which promotes the use of HTTP methods when building web application, instead of custom methods or protocols such as SOAP or WSDL. It defines several rules, including one that is similar to CRUD, which is described below.
HTTP has several methods that can perform processing on data as defined in the REST architecture, among which are 4 main methods. See this page:
http://en.wikipedia.org/wiki/HTTP#Request_methods
The table below offers a comparison with CRUD and SQL:

HTTP / REST	CRUD	SQL
POST	Create	INSERT
GET	Retrieve	SELECT
PUT	Update	UPDATE
DELETE	Delete	DELETE

Chapters in this tutorial

Chapters 1, 2 and 3 are mandatory, as they contain the fundamental knowledge for using the web service.
The next chapters, explores ways to interact with the web service using each of the REST operations, in order to give you the tools to make a full CRUD application.
If you only want to retrieve data, for example when developing a web application to notify you of orders, you might only be interested in Chapter 4.
If you prefer to develop a more complete application, chapters 4 to 7 will interest you.
Chapters 9 and 10 give you more detail on specific content management.
Finally, a cheat-sheet will give you quick hints and reminders.
Here are the chapters in this tutorial:

Chapter 1 - Creating an access to the back office

Before you can do anything, you first need to create an access to the web service.

Enabling the web service

Go in the PrestaShop back office, open the "Web service" page under the "Advanced Parameters" menu, and then choose "Yes" for the "Enable PrestaShop Webservice" option.
In previous versions of PrestaShop, you had to create a .htaccess file (or edit the existing one) in order to make your web service work.
This is no longer the case in PrestaShop 1.5+: the .htaccess will automatically be generated, while keeping your custom rules.

Creating your access

Open the "Web service" page under the "Advanced Parameters", and then click the "Add New" button to access the account configuration section. A long form appears:

- **Key**. Click the "Generate" button to generate an authentication key. You can also create your own (which must be 32 characters long), but using a generated key prevents wrong-doers from guessing your key.
  Using this key, you and other selected users will be able to access the web service.
- **Key description**. The description is not public, but make sure to put all the keywords pertaining to its use, so that you can find they key more quickly. For instance, a reminder of who that key is for, and what it gives access to.
- **Status**. You can disable any key at any time. This enables you to only temporarily grant access to your data from a certain key.T
- **Permissions**. This section is very important, as it enables you to assign rights for each resource you want to make available for this key. Indeed, you might want a user to have read and write access on some resources, but only read access on others – and no access to the more important ones.
  In the list of permissions, the left button allows you to define all the rights for a given resource. Select the resources you need to manipulate from your application; in our case check the first check box in the "customers" row and then press "Save".
- **Shop association**. This only appears in multistore mode: you can choose the shop to which the key has access.

If you define your own passkey, make sure it is very secure and that its rights are limited.

Chapter 2: Discovery: Testing your access to the web service with the browser

Accessing /api/

To test if you have properly configured your access to the web service, try to access your online shop with the following URL: http://mypasskey@example.com/api/, where mypasskey is replaced with the key you created, and example.com with your shop's URL. Mozilla Firefox is the preferred browser to test your access, but any browser able to display XML should do just fine.
The shop should prompt you for a username and a password to enter. The username is the authentication key you created and **there is no password to enter**.
Another method is to go directly to the following page of your shop: http://example.com/api/
Of course, it can also work locally: http://127.0.0.1/ps1541/api/ gives the same result.
The /api/ folder gives you access to the list of resources that you configured in your back office, with all the permissions you chose to grant. **If no permission has been set for the key, than the browser will keep asking you to enter the key indefinitely**. If you cannot access some resources, the API might answer with this XML response:

```
<?xml version="1.0" encoding="UTF-8"?>
<prestashop xmlns:xlink="http://www.w3.org/1999/xlink">
  <errors>
    <error>
      <message><![CDATA[Internal error. To see this error please display the PHP
errors.]]></message>
    </error>
  </errors>
</prestashop>
```

Using XLink, you will then be able to access your various resources. XLink associates an XML file to another XML file via a link.

Available resources

The /api/ URL gives you the root of all the resources, in the form of an XML file.
Here it is, extremely simplified. This list is current as of version 1.5.4.1 of PrestaShop. Note that we only show the API resources available for one of the available stores.

```
<?xml version="1.0" encoding="UTF-8"?>
<prestashop xmlns:xlink="http://www.w3.org/1999/xlink">
  <api shop_name="MYSHOP">
    <addresses>...</addresses>
    <carriers>...</carriers>
    <cart_rules>...</cart_rules>
    <carts>...</carts>
    <categories>...</categories>
    <combinations>...</combinations>
    <configurations>...</configurations>
    <contacts>...</contacts>
    <content_management_system>...</content_management_system>
    <countries>...</countries>
    <currencies>...</currencies>
    <customer_messages>...</customer_messages>
    <customer_threads>...</customer_threads>
    <customers>...</customers>
    <deliveries>...</deliveries>
    <employees>...</employees>
    <groups>...</groups>
    <guests>...</guests>
    <image_types>...</image_types>
    <images>...</images>
    <languages>...</languages>
    <manufacturers>...</manufacturers>
    <order_carriers>...</order_carriers>
    <order_details>...</order_details>
    <order_discounts>...</order_discounts>
    <order_histories>...</order_histories>
    <order_invoices>...</order_invoices>
    <order_payments>...</order_payments>
    <order_states>...</order_states>
    <orders>...</orders>
    <price_ranges>...</price_ranges>
    <product_feature_values>...</product_feature_values>
    <product_features>...</product_features>
    <product_option_values>...</product_option_values>
    <product_options>...</product_options>
    <product_suppliers>...</product_suppliers>
    <products>...</products>
    <search >...</search>
    <shop_groups>...</shop_groups>
    <shops>...</shops>
    <specific_price_rules>...</specific_price_rules>
    <specific_prices>...</specific_prices>
    <states>...</states>
    <stock_availables>...</stock_availables>
    <stock_movement_reasons>...</stock_movement_reasons>
    <stock_movements>...</stock_movements>
```

```
    <stocks>...</stocks>
    <stores>...</stores>
    <suppliers>...</suppliers>
    <supply_order_details>...</supply_order_details>
    <supply_order_histories>...</supply_order_histories>
    <supply_order_receipt_histories>...</supply_order_receipt_histories>
    <supply_order_states>...</supply_order_states>
    <supply_orders>...</supply_orders>
    <tags>...</tags>
    <tax_rule_groups>...</tax_rule_groups>
    <tax_rules>...</tax_rules>
    <taxes>...</taxes>
    <translated_configurations>...</translated_configurations>
    <warehouse_product_locations>...</warehouse_product_locations>
    <warehouses>...</warehouses>
    <weight_ranges>...</weight_ranges>
    <zones>...</zones>
  </api>
  <api shop_name="MYOTHERSHOP">...</api>
  <api shop_name="YETANOTHERSHOP">...</api>
</prestashop>
```

As with any XLink-bearing XML file, each resource element leads to more resources, linked from the xlink attribute.

```
<customers xlink:href="http://example.com/api/customers" get="true" put="true"
post="true" delete="true" head="true">
  <description xlink:href="http://example.com/api/customers" get="true" put="true"
post="true" delete="true" head="true">The e-shop's customers</description>
  <schema xlink:href="http://example.com/api/customers?schema=blank" type="blank"/>
  <schema xlink:href="http://example.com/api/customers?schema=synopsis"
type="synopsis"/>
</customers>
```

In addition, the element contains a description of the resource, and two schemas: a blank one, which you can use to create a new item, and a synopsis one, which is just like the blank one but with a detailed description of what type of data is expected in each element.
Here is an extract of the Customer blank schema:

Blank schema

```
<?xml version="1.0" encoding="UTF-8"?>
<prestashop xmlns:xlink="http://www.w3.org/1999/xlink">
  <customer>
    <id></id>
    <id_default_group></id_default_group>
    <id_lang></id_lang>
    <newsletter_date_add></newsletter_date_add>
    <ip_registration_newsletter></ip_registration_newsletter>
    <last_passwd_gen></last_passwd_gen>
    <secure_key></secure_key>
    <deleted></deleted>
    <passwd></passwd>
    <lastname></lastname>
    <firstname></firstname>
    <email></email>
    ...
  </customer>
</prestashop>
```

Here is an extract of the Customer synopsis schema:

Synopsis schema

```
<?xml version="1.0" encoding="UTF-8"?>
<prestashop xmlns:xlink="http://www.w3.org/1999/xlink">
  <customer>
    <id_default_group></id_default_group>
    <id_lang format="isUnsignedId"></id_lang>
    <newsletter_date_add></newsletter_date_add>
    <ip_registration_newsletter></ip_registration_newsletter>
    <last_passwd_gen></last_passwd_gen>
    <secure_key format="isMd5"></secure_key>
    <deleted format="isBool"></deleted>
    <passwd required="true" maxSize="32" format="isPasswd"></passwd>
    <lastname required="true" maxSize="32" format="isName"></lastname>
    <firstname required="true" maxSize="32" format="isName"></firstname>
    <email required="true" maxSize="128" format="isEmail"></email>
    ...
  </customer>
</prestashop>
```

The value types can be found in the Web service reference.

Navigating your data

In this tutorial, we are going to base our examples on the management of customers through the API, but you can use just about any data
In the Customers element, you should get these attributes and tags:

```
<customers xlink:href="http://example.com/api/customers" get="true" put="true"
post="true" delete="true" head="true">
  <description xlink:href="http://example.com/api/customers" get="true" put="true"
post="true" delete="true" head="true">The e-shop's customers</description>
  <schema xlink:href="http://example.com/api/customers?schema=blank" type="blank"/>
  <schema xlink:href="http://example.com/api/customers?schema=synopsis"
type="synopsis"/>
</customers>
```

The get, put, post, and delete attributes have the value true, meaning that you have correctly configured the customers resource for the current, and that their data is accessible.
You can now use the XLink link that shows up on the URL http://example.com/api/customers and go to it:
Result for http://example.com/api/customers

```
<?xml version="1.0" encoding="UTF-8"?>
<prestashop xmlns:xlink="http://www.w3.org/1999/xlink">
  <customers>
    <customer id="1" xlink:href="http://example.com/api/customers/1"/>
    <customer id="2" xlink:href="http://example.com/api/customers/2"/>
    <customer id="3" xlink:href="http://example.com/api/customers/3"/>
    <customer id="4" xlink:href="http://example.com/api/customers/4"/>
    <customer id="5" xlink:href="http://example.com/api/customers/5"/>
    <customer id="6" xlink:href="http://example.com/api/customers/6"/>
    <customer id="7" xlink:href="http://example.com/api/customers/7"/>
    ...
  </customers>
</prestashop>
```

Then, from the customers list which is displayed via http://example.com/store/api/customers, you can access the XLink corresponding to each customer. This gives you all the information
Result for http://example.com/api/customers/1

<?xml version="1.0" encoding="UTF-8"?>
<prestashop xmlns:xlink="http://www.w3.org/1999/xlink">
  <customer>
    <id><![CDATA[1]]></id>
    <id_default_group
xlink:href="http://example.com/api/groups/3"><![CDATA[3]]></id_default_group>
    <id_lang xlink:href="http://example.com/api/languages/1"><![CDATA[1]]></id_lang>
    <newsletter_date_add><![CDATA[2013-04-23 11:40:52]]></newsletter_date_add>
    <ip_registration_newsletter></ip_registration_newsletter>
    <last_passwd_gen><![CDATA[2013-04-23 05:40:52]]></last_passwd_gen>
    <secure_key><![CDATA[5b402973df21cd42a303a2fdfce91df7]]></secure_key>
    <deleted><![CDATA[0]]></deleted>
    <passwd><![CDATA[a747961318242111b33ed551dcff10a9]]></passwd>
    <lastname><![CDATA[DOE]]></lastname>
    <firstname><![CDATA[John]]></firstname>
    <email><![CDATA[pub@prestashop.com]]></email>
    <id_gender><![CDATA[1]]></id_gender>
    <birthday><![CDATA[1970-01-15]]></birthday>
    <newsletter><![CDATA[1]]></newsletter>
    <optin><![CDATA[1]]></optin>
    <website></website>
    <company></company>
    <siret></siret>
    <ape></ape>
    <outstanding_allow_amount><![CDATA[0.000000]]></outstanding_allow_amount>
    <show_public_prices><![CDATA[0]]></show_public_prices>
    <id_risk><![CDATA[0]]></id_risk>
    <max_payment_days><![CDATA[0]]></max_payment_days>
    <active><![CDATA[1]]></active>
    <note></note>
    <is_guest><![CDATA[0]]></is_guest>
    <id_shop><![CDATA[1]]></id_shop>
    <id_shop_group><![CDATA[1]]></id_shop_group>
    <date_add><![CDATA[2013-04-23 11:40:52]]></date_add>
    <date_upd><![CDATA[2013-04-23 11:40:52]]></date_upd>
    <associations>
      <groups node_type="group">
        <group xlink:href="http://example.com/api/groups/3">
          <id><![CDATA[3]]></id>
        </group>
      </groups>
    </associations>
  </customer>
</prestashop>

Adding and editing a resource

The http://example.com/api/customers/1 example is not only available using the HTTP GET method, but also using POST, PUT, DELETE, HEAD.
To add a customer (or any other resource, for that matter), you simply need to GET the XML blank data for the resource (/api/customer?schema=blank), fill it with your changes, and POST the whole XML file to the /api/customers/ URL again. PrestaShop will take care of adding everything in the database, and will return an XML file indicating that the operation has been successful, along with the ID of the newly created customer.
To edit an existing resource: GET the full XML file for the resource you want to change (/api/customers/7), edit its content as needed, then PUT the whole XML file back to the same URL again.

Chapter 3 - First steps: Accessing the web service and listing customers

Preparation

1. Configure your PHP installation so that it has the cURL extension installed and activated:
    - With Windows: Place this line in your php.ini file:

○ extension=php_
curl.dll

    - Linux/Mac: install the cURL extension:

○ sudo apt-get install
php5-curl

2. Copy the PSWebServiceLibrary.php file at the root of your Web server. You can download it from this URL: https://github.com/PrestaShop/PrestaShop-webservice-lib/archive/master.zip
3. You can also do this tutorial on a local server even while your shop is on the Internet.
4. Create a list_the_customers.php file at the root of the Web server that you have chosen.
5. Specify where to find the web server in your file:

6. require_once('./
PSWebServiceLibrary.php');

Configured this way, your file should be found in the same folder as PSWebServiceLibrary.php.

Accessing the web service

In this section we will see how to access the web service using the PHP library.
First, you must create an instance of the PrestaShopWebservice object, which takes 3 parameters in its constructor:

- The store's root path (ex: http://example.com/).
- The authentication key (ex: ZR92FNY5UFRERNI3O9Z5QDHWKTP3YIIT).
- A boolean value, indicating whether the Web service must use its debug mode.

If you do not understand the terms of object-oriented programming such as instance, method, or constructor, that's okay for the rest of the tutorial. Here's how you create a Web service call:

```
$webService = new PrestaShopWebservice('http://example.com/',
'ZR92FNY5UFRERNI3O9Z5QDHWKTP3YIIT', false);
```

Once the instance is created, you can access the following methods:

Method	HTTP equivalent
get()	GET
add()	POST
edit()	PUT
delete()	DELETE

We will explain how to use of these methods in other parts of the tutorial.

Handling errors

It is essential that you understand how to handle errors with the web service library. By implementing error-catch method early, you will more easily detect issues, and be able to correct them on the go.
Error handling with the web service library is done using PHP exceptions. If you do not know about them, you should read http://php.net/manual/en/language.exceptions.php, as exceptions are an essential part of good coding practice.

How it works

The error handling is done within a try..catch block, with the web service processing being done in the try section, the catch one containing the error handling code.

```
try {
  // Execution ( if an error occurs in this code, stops and goes in the catch block)
}
catch {
  // Error handling (tries to catch the error or the error display)
}
```

Example

That means each creation or use of the library must be located within a "try" block. The "catch" block can then handle the error if it occurs during the execution of the try block.
Now we'll see how to list all customers via the web service, and then we will see the four CRUD methods.
In the following code sample, we want to to get the list of all customers:

1. In the try block, we instantiate the PrestaShopWebservice object with the necessary parameters and retrieve the customer resource XML in a variable.
2. In the catch block, we put code to display the PHP error message, if anything wrong happens in the try block.

```
try {
    // creating web service access
    $webService = new PrestaShopWebservice('http://example.com/',
'ZR92FNY5UFRERNI3O9Z5QDHWKTP3YIIT', false);

    // call to retrieve all customers
    $xml = $webService->get(array('resource' => 'customers'));
}
catch (PrestaShopWebserviceException $ex) {
    // Shows a message related to the error
    echo 'Other error: <br />' . $ex->getMessage();
}
```

Listing customers

Let's now see how to view a full list of customer IDs. We could display more information and customize it, but that's for another part of this tutorial
As we saw in the previous code sample, we need the get() method to retrieve an XML file containing all the customers. The parameter has to be a key-value array, where we define the resource we want:

Key	Value
resource	customers

```
// The key-value array
$opt['resource'] = 'customers';

Retrieving the XML data
$xml = $webService->get($opt);
```

The value defines the resource that the web service will use in a future call. The value could be carrier types, countries or any other type of resource that can be found in the "Web service" menu of your back office.

Result

Launching the code above will return a SimpleXML object containing all the customer IDs.

```
<?xml>
<prestashop>
  <customers>
    <customer>
      customer ID
    </customer>
    <customer>
      customer ID
    </customer>
    <customer>
      customer ID
    </customer>
 ...Other customer tags
  </customers>
</prestashop>
```

Now we need to access the tags that interest us in the XML file.

Structure

The data returned by calling $webService->get puts us at the root of the document.
To access the fields of customers who are children from the Customers tag, we only need to retrieve all fields in an associative array in SimpleXML like this:

```
$resources = $xml->customers->children();
```

From there, we can access customer IDs easily. Here's an example with a path from identifiers:

```
foreach ($resources as $resource)
    echo $resource->attributes() . '<br />';
```

Thanks to these elements, we can create a HTML table containing all the customer IDs. Try that before reading the next chapter.
You can use the "Customers" menu in the back office to find the IDs of all customers. If you encounter difficulties, have a look at the example file named 0-CustomersList.php, to see the results you should get.

Chapter 4 - Data retrieval: Retrieving a customer

Goal: List and display information from a customer.
Difficulty: *
Problem: How to create a system that allows customers using IDs to retrieve customer records?

Preparation

Duplicate the file list_the_customers.php that was created in the previous chapter, rename it to R-CRUD.php, and put it at the root of your Web server.
If you didn't finish the previous chapter, duplicate the file 0-CustomersList.php and rename it R-CRUD.php.
In the XML data that we retrieved, which contains the list of customers, we will find all the XLinks providing access to customer information.

```
<customers>
    <customer id="1" xlink:href="http://example.com/api/customers/1" />
</customers>
```

Example

The XLink for the customer tag with ID 1 is: http://example.com/api/customers/1
This link retrieves a new XML file, which contains information about the customer who has ID 1.
In order to manage access to different customers, we will associate page identifiers with customers via a GET parameter named "id". The link http://example.com/R-CRUD.php?id=1 we will display the file for customer 1.
We must change the table in the file created in the previous chapter to add a link to future customer files.
We will have to isolate the display from the display list of a particular customer.
In order to do this, we must isolate the display of the list by checking that the "id" GET parameter property is not present when viewing your list. We do this using isset().
Calling the web service is exactly the same as displaying the list, except that if you need to add the ID element to the table whose value is the id of a customer.
Currently, we are using the customers or clients resources. If we'd been trying to change the countries resources, this ID would have been a country ID.

```
$opt['resource'] = 'customers';
$opt['id'] = $_GET['id'];
$xml = $webService->get($opt);
```

Use isset() before setting an ID enables you to easily carry out everything in this chapter.
Accessing resources is performed as above for displaying the list, because the tags that interest us are children of the customers tag.

```
$resources = $xml->customers->children();
```

This path can be created in another way (here in an HTML table):

```
foreach ($resources as $key => $resource)
    echo 'Name of field: ' . $key . ' - Value: ' . $resource . '<br />';
```

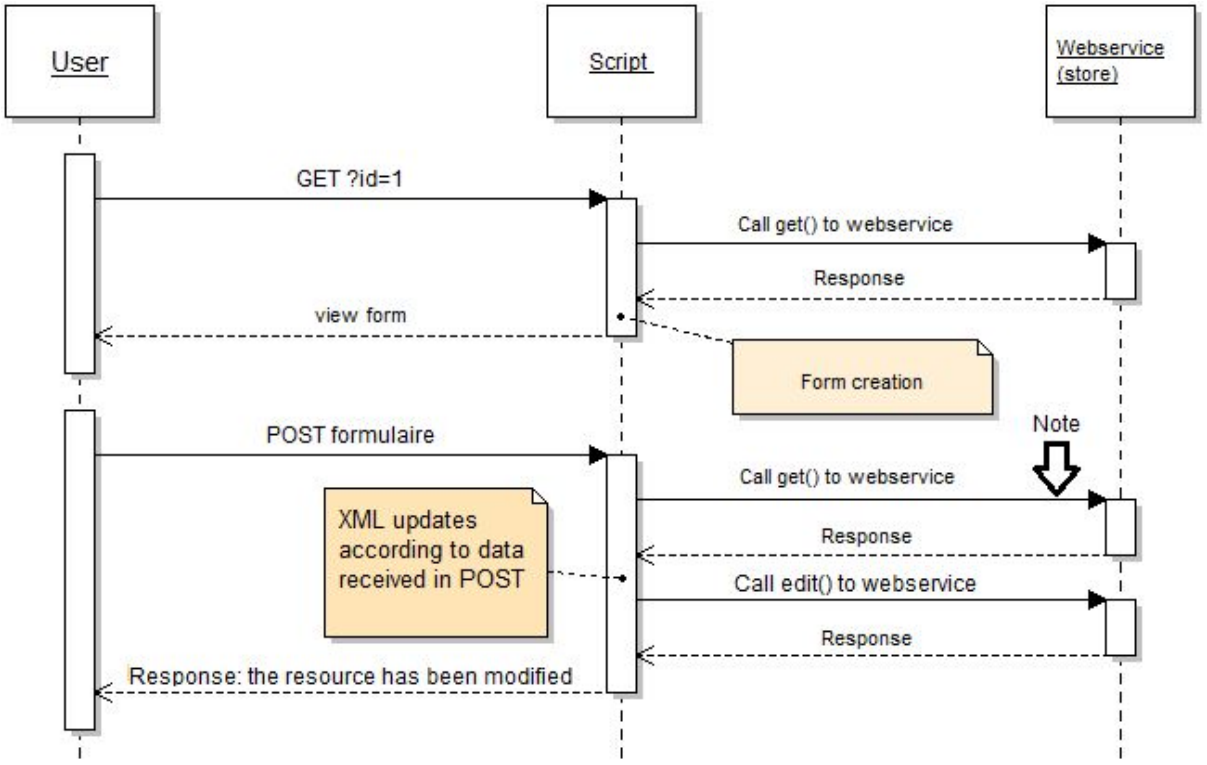
You now have everything needed to create a script to both list and display information for a particular customer.
Try creating this R-CRUD.php script. If you encounter any difficulties, follow the example from the 1-Retrieve.php file, which contains the result to which you should get.
In another chapter, we will see how to filter, sort and limit the number of items displayed in the list. If you're in a hurry to implement these features, you can find more information in Chapter 8.

Chapter 5 - Data modification: Updating a customer

Goal: A Web application to list and update customer information.
Difficulty: ***

Preparation

Duplicate file list_the_customers.php from Chapter 3.3, rename it to U-CRUD.php, and put it at the root of your Web server.
Updating resources via the web service is complex, so we will first explain its operation.
Sequence diagram representing how a resource is updated:



We can see that the diagram is divided into 2 parts:

1. Getting the resource to a defined id (1 in the diagram) and creating of the form.
2. Update resource.

The arrow points to a get(), which means a resource retrieval.
This step is important because we need to get the XML file back in order to match it with the data sent by the form before we can call "edit" to update the resource.
Note that we could have modified it otherwise by sending an XML using JavaScript, and thus have not used get() in this process.

Step 1: Getting data and form creation

Retrieving the XML file and display the form:

```
// Define the resource
$opt = array('resource' => 'customers');

// Define the resource id to modify
$opt['id'] = $_GET ['id'];

// Call the web service, recuperate the XML file
$xml = $webService->get( $opt );

// Retrieve resource elements in a variable (table)
$resources = $xml->children()->children();

// customer form
```

Here, the call is similar to data retrieval. It is this call that will enable us to create the form. We will generate the automatic update form.
For now, use HTML tags "input" as having as its "name" the name of the attribute, and as its "value" the value of the attribute.
In order not to lose the id for the second step according to the diagram, the form will show up as: ?id = "Customer Id"
Thus we will get it this way:

```
$_GET['id'];
```

We could have done this differently, such as by passing this ID in a POST request, but you will see that this method will simplify the processing that follows.

Step 2: Update the resource

Initially, as you can see from the "Note" arrow in the diagram, we will retrieve the XML file. For this, you will carry out the same call as you did when you created the form.
If you have specified, as indicated above, the form destination with an id, your call should already be done and the form will be redisplayed.
Help for creating a form:

```
foreach ($resources as $key => $resource) {
    echo '<tr><th>' . $key . '</th><td>';
    echo '<input type="text" name="' . $key . '" value="' . $resource . '"/>';
    echo '</td></tr>';
}
```

Once the XML file is retrieved we need to modify the new data with data received by POST. Path of the keys in the XML file and update values:

```
foreach ($resources as $nodeKey => $node) {
    $resources->$nodeKey =
$_POST[$nodeKey];
}
```

We now have an XML file updated. Now we just need to send it.
Example of an update:

```
// Resource definition
$opt = array('resource' => 'customers');

//XML file definition
$opt['putXml'] = $xml->asXML();

// Definition of ID to modify
$opt['id'] = $_GET[ 'id' ];

// Calling asXML() returns a string corresponding to the file
$xml = $webService->edit($opt);
```

Now, in your U-CRUD.php script, try to modify a customer with an ID defined in the code, then do it for all customers.
Check using R-CRUD.php that the information has been changed and then process the customer ID.
If you have trouble, look at the code in the 2-update.php sample file.

Chapter 6 - Data creation: Creating a remote online form

Creation - Remote Online Form

Objective: A Web application to list and create a new customer.
Difficulty: **

Preparation

Copy the file list_the_customers.php from Section 3.3 to a file named C-CRUD.php at the root of your Web server.
The addition of resource can be likened to an upgrade from an empty element.
But how do we retrieve an XML formatted as an empty customer?
In the web service, there is a method to retrieve an empty XML. It is accessible via a URL formatted as follows: http://example.com/api/(resource name)?schema=blank
It is possible to replace the parameter scheme value "blank" with "synopsis" in order to gain more information about the resource fields.
As we saw in Section 3.3 (List customers) it is possible make an array of parameters for "get ", "resource," and "id." It is also possible to specify only one URL this way:

```
$xml = $webService->get(array('url' =>
'http://example.com/api/customers?schema=blank'));
```

Here, we get the entire XML variable from an empty customer.

```
<prestashop>
  <customer>
    <id_default_group/>
etc...Beginning of the XML file retrieved:
```

We can then, thanks to the many fields we have, create an associated form.

Getting of all fields

```
$resources = $xml->children()->children();
```

Path of all fields and part of the dynamic creation of form fields in a table

```
foreach ($resources as $key => $resource) {
    echo '<tr><th>' . $key . '</th><td>';
    echo '<input type="text" name="' . $key . '" value=""/>';
    echo '</td></tr>';
}
```

Once the data is passed in POST, we combined the data sent with the blank XML file, which is the same technique used for updating data.

```
foreach ($resources as $nodeKey => $node) {
    $resources->$nodeKey =
$_POST[$nodeKey];
```

```
}
```

Calling the web service is similar to what we have seen previously:

```
$opt = array('resource' => 'customers');
$opt['postXml'] = $xml->asXML();
$xml = $webService->add($opt);
```

Now create a script that adds a customer. Remember that some fields are mandatory, so do not forget to fill them out.
If you have trouble, look at the code the 3-Create.php sample file.
When a customer is created from within PrestaShop's administration interface, a confirmation e-mail is sent to the customer. This cannot be done directly with the webservice: there is no way to trigger the sending of that confirmation e-mail.
However, you can create an override file for the Customer class and override the addWs() method. This method is similar to ObjectModel::add() but is only called from the webservice. You can find examples of its use in the Product and Order classes.

Chapter 7 - Data removal: Removing customer accounts from the database

Objective: A Web application for listing and deleting customers.
Difficulty: *

Preparation

Duplicate file list_the_customers.php from Chapter 3, rename it to D-CRUD.php and put it at the root of your Web server.
For this last part of our tutorial, we will learn how to delete a resource.
Here is the complete, detailed code you need in order to remove a customer:

```
try {
    // Create an instance
    $webService = new PrestaShopWebservice('http://example.com/' ,
'ZR92FNY5UFRERNI3O9Z5QDHWKTP3YIIT' , false);
    $opt['resource'] = 'customers';                 // Resource to use
    opt['id'] = 3;                                  // ID to use
    $webService->delete($opt);                      // Delete
    // If we can see this message, that means we have not left the try block
    echo 'Customer '.opt['id'].' successfully deleted!';
}
catch (PrestaShopWebserviceException $ex) {
    $trace = $ex->getTrace();              // Retrieve all info on this error
    $errorCode = $trace[0]['args'][0];     // Retrieve error code
    if ($errorCode == 401)
        echo 'Bad auth key';
    else
        echo 'Other error: <br />'.$ex->getMessage();
    // Display error message{color}
}
```

This code enables you to remove a customer whose ID is "3". As you can see, deleting the customer differs only slightly from retrieving a resource. In fact the only thing different in the code lies in the method called: We will no longer call this method get() but instead simply delete()!
You must now replace the customer ID by a dynamically-defined ID.
Now create all the script that will display a list of customer IDs and delete a customer of your choice.
Again, if you have trouble, look at the code from the 4-delete.php sample file.

Chapter 8 - Advanced use

Here are a few sample advanced uses.

Rendering Options

Include all fields from the "products" resource

URL: (Store URL)/api/products/?display=full
PHP:

```
$opt = array(
    'resource' => 'products',
    'display'  => 'full'
);
```

Include only the ID of all carriers

URL: (Store URL)/api/carriers/?display=[id]
PHP :

```
$opt = array('resource' => 'carriers', 'display' => '[id]');
```

Only include the "name" and "value" fields from the "configurations" resource

URL: (Store URL)/api/configurations/?display=[name,value]
PHP:

```
$opt = array(
    'resource' => 'configurations',
    'display'  => '[name,value]'
);
```

Rendering Filters

Only include the first and last names of customers "customers" whose ids are 1 or 5

URL: (Store URL)/api/customers/?display=[firstname,lastname]&filter[id]=[1|5]
PHP:

```
$opt = array(
    'resource'   => 'customers',
    'display'       => '[firstname,lastname]',
    'filter[id]' => '[1|5]'
);
```

Only include the last names of customers "customers" whose ids are between 1 and 10

URL: (Store URL)/api/customers/?display=[lastname]&filter[id]=[1,10]
PHP:

```
$opt = array(
    'resource'   =>'customers',
    'display'       => '[lastname]',
    'filter[id]' => '[1,10]'
);
```

Only include the birthday of clients whose name is "John" and whose last name is "Doe"

URL: (Store URL)/api/customers/?display=[birthday]&filter[firstname]=[John]&filter[lastname]=[DOE]
PHP:

```
$opt = array(
    'resource'              =>'customers',
    'display'               => '[birthday]',
    'filter[firstname]' => '[John]',
    'filter[lastname]'  => '[DOE]'
);
```

Only include the names of manufacturers "manufacturers" whose name begins with "Appl"

URL: (Store URL)/api/manufacturers/?display=[name]&filter[name]=[appl]%
PHP:

```
$opt = array(
    'resource'    => 'manufacturers',
    'display'       => '[name]',
    'filter[name]' => '[appl]%'
);
```

Sorting Filters

Filter the customers "customers" in alphabetical order according to last name

URL: Store URL/api/customers?display=full&sort=[lastname_ASC]
PHP:

```
$opt = array(
    'resource' => 'customers',
    'display'  => 'full',
    'sort'          =>
'[lastname_ASC]'
);
```

Filters to limit rendering

Only include the first 5 states "states"

URL: (Store URL)/api/states/?display=full&limit=5
PHP:

```
$opt = array(
    'resource' => 'states',
    'display'  => 'full',
    'limit'=> '5'
);
```

Only include the first 5 elements starting from the 10th element from the states resource "states"

URL: (Store URL)/api/states/?display=full&limit=9,5
PHP:

```
$opt = array(
    'resource' => 'states',
    'display'  => 'full',
    'limit'=> '9,5'
);
```

Chapter 9 - Image management

Accessing the images is done through the images entity.
Several types of images are available.

- General shop images
- Product images
- Category images
- Customization images
- Manufacturer images
- Supplier images
- Store images

They can be reach using the following links:

- /api/images/general
- /api/images/products
- /api/images/categories
- /api/images/customizations
- /api/images/manufacturers
- /api/images/suppliers
- /api/images/stores

Various image size are available, depending on the image types. They are available as XLink links in the links above, encapsulated in the image_types node.
For instance, in order to retrieve the image with the id 10 for the product with id 5, we would use the following path: /api/images/products/5/10.

Changing the images

In order to change the available images, we have to use a POST request, with the new image as its parameter.
For instance, here is how to **change** the image for category 2:

- HTTP method: POST (/!\ not PUT)
- URL: /images/categories/2
- POST content: [binary content for the new image]

...and here is how to **add** a new image to the product with id '1':

- HTTP method: POST
- URL: /images/products/1
- POST content: [binary content for the new image]

Here is an HTML form enabling images to be sent:
Adding new image

```
<form enctype="multipart/form-data" method="POST"
action="http://eWURcnNJ6mbLUHB10EEIzTZsTShs33IX@myprestahop.com/api/images/products/1">
  <fieldset>
    <legend>Add image for products No 1</legend>
    <input type="file" name="image">
    <input type="submit" value="Execute">
  </fieldset>
</form>
```

Update existing image

```
<form
action="http://eWURcnNJ6mbLUHB10EEIzTZsTShs33IX@myprestahop.com/api/images/products/1/2" method="POST" enctype="multipart/form-data">
<fieldset>
    <legend>Update product image 1/2</legend>
    <input name="ps_method" value="PUT" type="hidden">
    <input name="image" type="file">
    <input value="Execute" type="submit">
</fieldset>
</form>
```

If you would rather use cURL:

```
$url = 'http://myprestashop.com/api/images/products/1';
/**
 * Uncomment the following line in order to update an existing image
 */
//$url = 'http://myprestashop.com/api/images/products/1/2?ps_method=PUT';

$image_path = 'C:\\my_image.png';
$key = 'My web service key';

$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $url);
curl_setopt($ch, CURLOPT_POST, true);
curl_setopt($ch, CURLOPT_USERPWD, $key.':');
curl_setopt($ch, CURLOPT_POSTFIELDS, array('image' =>
'@'.$image_path));
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$result = curl_exec($ch);
curl_close($ch);
```

Chapter 10 - Price management

Description

It is possible to retrieve the customer prices for a given product, simply by passing parameters to a web service query.
These custom prices are available for the following entities:

- products
- combinations

Custom prices have an alias that you must indicate.

Example

Let's say we want to retrieve the price for combination '25' of the product '1', with tax, in a web service field name "my_price".
The query:
/api/products/1?price[my_price][use_tax]=1&price[my_price][product_attribute]=25
Before PrestaShop 1.6.0.10, this query retrieved the price **without** tax. The behavior has been changed to make it more on par with needs.

Parameters

Here is the list of available parameters when needing to define a custom price:

- country
- state
- currency
- group
- quantity
- decimals
- product_attribute
- use_tax
- use_reduction
- use_ecotax
- only_reduction

Cheat-sheet - Concepts outlined in this tutorial

Summary of the available methods in the library

To help you get started with web service, here's a little memo of techniques used in this tutorial.

			Method parameters			
C	POST	add	X	X		X
D	DELETE	delete	X	X	X	
R	GET	get	X	X	X	
U	UPDATE	edit	X	X	X	X
	REST	**Method**	**url**	**resource**	**id**	**xml**

If the URL parameter is specified, no other setting can be used and vice versa.

Options

Key	Value	Description
output_format	XML, JSON	Change the output format
ps_method	GET, POST, PUT, DELETE	Override the HTTP method used for the request
display	[field1,field2 ...]	Only display fields in brackets
display	full	Display all fields

Key	Key suffix	prefix	Value	Suffix	Description
filter	[field]		[value1	[value2]	Filter "field" with value between "value1" and "value2"
filter	[field]		[value]		Filter field with the value "value"
filter	[field]		[value1,value2...]		Filter fields for values specified between brackets
filter	[field]	%	[value]	%	Filter "columns" for values containing "value"

Key	Value	Description
sort	[field1_ASC,field2_DESC,field3_ASC]	Sort by field with the suffix _ASC _DESC or in the order
sort	full	show all fields

Key	Value	Description
limit	Number	Limit the result to "Number"
limit	Starting index, Number	Limit the result to "Number" from the "Index"

Using the library in multistore mode

In order to use web services in when the multistore feature is enabled, you simply have to add the id_shop parameter.
The shops entity enables you to access to the list of shops as well as their associated identifiers.

Key	Value	Description
id_shop	Shop ID	Define the shop to be used as a context for the web service.
id_group_shop	Group shop ID	Define the group shop to be used as a context for the web service.