

Flex and PHP

Mobile is big. The web is big. Services are big. It is no secret that in the past several years the API has become king. Simply accessing a web site is not sufficient any more.

You need to be able to access the web site, and its data, in a manner that makes it useful to other applications. Traditional web development just doesn't cut it any more. What do I mean? I'm glad you asked. The *traditional* way of developing for multiple destinations would be something like this.

```
<html>
<head><title>Example</title></head>
<body>
<?php
$select = $conn->select()
->from('peaks_location', 'range')
->where('state = ?', $state)
->group('range')
->order('range');
$res = array();
foreach ($select->query()->fetchAll() as $row) {
echo $row['range'] . '<br />';
}
}
```

That code would handle the regular browser request. And then you'd have another file or another controller where you'd have this code to handle JSON-RPC:

```
header('content-type: application/json');
$select = $conn->select()
->from('peaks_location', 'range')
->where('state = ?', $state)
->group('range')
->order('range');
$res = array(
'jsonrpc' => '2.0',
'result' => array()

foreach ($select->query()->fetchAll() as $row) {
$res['result'][] = $row['range'];
}
echo json_encode($res);
```

Easy to do. However, it is horribly un-maintainable. What happens if there is a change in the logic? Then you need to find all the places where you implemented this and change each one. But since you want to make it available in two formats what other option do you have?

What we're really trying to do in this whole exercise is write an application that can be used equally for browser, JavaScript, AMF or remote server implementations. But what we want to do is write the interface just once. The key to the whole application is in the directory /applications/mappers (see the downloaded workspace). What we're doing here is following the Data Mapper Design Pattern. While it may be tempting to build your application by putting your logic in your controller or in a PHP script, wrapping all of your functionality in a mapper allows you to easily create multiple access points for individual pieces of functionality. That's where the mapper comes in. A mapper is simply a common piece of functionality that can be accessed from multiple places. This is usually done as an object. Following is an example of what this class definition would look like.

```
class Mapper_Peaks
{
public function getMountains($state, $range)
{
/* TODO */
}
}
```

Here we have omitted the functionality but you can see the structure. The code that we had previously defined would then be placed in the getRanges() method. That way you could simply create

```
<html>
<head><title>Example</title></head>
```

```

<body>
<?php
$mapper = new Mapper_Peaks();
foreach ($mapper->getRanges($state) as $range) {
echo $range . '<br />';
}
OR
header('content-type: application/json');
$mapper = new Mapper_Peaks();
foreach ($mapper->getRanges($state) as $range) {
$res['result'][] = $range;
}
echo json_encode($res);

```

There's also the additional benefit of the mapper code being MUCH easier to unit test. Easy enough, so we're done, right?

Not really. What if you also needed to make it available via AMF (shocking, I know). Then you'd need a third section of code. So while we're doing better we're not quite where we need to be.

Thankfully, there are a lot of things that Zend Framework can be used for. Additionally, Zend Framework does its absolute best to maintain consistent interfaces which means that you can implement as many of the interfaces as you need via the same method calls. So what we're going to do here is implement the interfaces using the Zend_Server implementation for JSON, AMF and XML. We used an MVC-based implementation here simply because MVC implementations are usually more *hookable*. In other words, you can hook into them a little easier.

The hook is a plugin we built which hooks into the Zend Framework MVC request lifecycle. (For more information you can see <http://www.eschrade.com/page/zend-framework-request-lifecycle-4b9a4288>). We are going to implement the `routeShutdown()` hook which is called immediately after the routing is done, but prior to the routes being executed. The reason why we put it there is because it allows us to intercept a request and route it to a different location if the request is a service request as opposed to a regular HTTP request see (Listing 1 below).

Listing 1.

```

class Ctx_Controller_Plugin_ServicePlugin extends Zend_Controller_Plugin_Abstract
{
    public function routeShutdown(Zend_Controller_Request_Abstract $request)
    {
        if ($request->isXmlHttpRequest () && $request->isPost ()) {
            $serviceHandler = new Zend_Json_Server ();
            $serviceHandler->getServiceMap ()->setDojoCompatible ( true );
        } else if (strpos ( $this->request->CONTENT_TYPE, 'application/x-amf' ) === 0 ) {
            $serviceHandler = new Zend_Amf_Server ();
        } else if (strpos ( $request->CONTENT_TYPE, 'text/xml' ) === 0 ) {
            $serviceHandler = new Zend_XmlRpc_Server (
                'http://' . $_SERVER ['HTTP_HOST'] . $_SERVER ['SCRIPT_NAME']
            );
        } else if ($this->_request->isXmlHttpRequest () ) {
            $serviceHandler = new Zend_Json_Server ();
            $smd = $serviceHandler->getServiceMap ();
            $smd->setDojoCompatible ( true );
            $request->setParam ( 'smd', $smd );
            $serviceHandler->setTarget ( $this->view->url () )
                ->setEnvelope ( Zend_Json_Server_Smd::ENV_JSONRPC_2 );
        }
        if ($serviceHandler) {
            foreach ( glob ( APPLICATION_PATH . '/mappers/*.php' ) as $dir ) {
                $name = substr ( basename ( $dir ), 0, - 4 );
                $class = 'Mapper_' . $name;
                $serviceHandler->setClass ( $class, $name );
            }
            $request->setParam ( 'serviceHandler', $serviceHandler );
            $request->setControllerName ( 'index' );
            $request->setActionName ( 'service' );
        }
    }
}

```

It seems like a lot of code but it's really not that bad. Here's a break down of the statements

- Is it an XMLHTTP Request and is it a POST? Create the Json server
- Is it an AMF request? Create the AMF server
- Is it an XmlRpc request? Create the XmlRpc server
- Is it an XMLHTTP Request and is it a GET? Create the Service Map (for JSON-RPC 2.0)
- If a service handler has been created add all of the application's mappers, attach the service handler to the request and redirect to the service action. Then in our service action, which we've redirected the request to, we have the following code see (Listing 2).

Listing 2

```
class IndexController extends Zend_Controller_Action
{

    public function serviceAction()
    {
        $serviceHandler = $this->getRequest()->getParam('serviceHandler');
        if ($serviceHandler instanceof Zend_Server_Interface) {
            $smd = $this->_request->getParam('smd');
            if ($smd) {
                echo $smd;
                exit;
            }

            echo $serviceHandler->handle();
            exit;
        }
        throw new Zend_Controller_Response_Exception('Not Found', 404);
    }

    public function indexAction()
    {
        $mapper = new Mapper_Peaks();
        $this->view->states = $mapper->getStates();
    }

    public function rangesAction()
    {
        /* TODO */
    }

    public function mountainsAction()
    {
        /* TODO */
    }

    public function jsonrpcAction()
    {
    }

}
```

The `serviceAction()` method is where every servicebased request will be routed. The reason for `$smd` is because as part of the JSON-RPC 2.0 protocol there needs to be some kind of service definition, similar to a WSDL file for Soap. So what we do is make the assumption that if a request is an XMLHttp request, but a GET, that it is a request for the SMD definition. We can make that assumption here simply because we standardized the JSON calls on JSON-RPC 2.0. If we hadn't we'd need to add an extra line or two to validate that the request was actually for the SMD. Beyond that, all we need to do is call the `handle()` method for any service request that is made and whichever service handler was attached to the request will be executed.

The other methods (`indexAction()`, `rangesAction()`, `mountainsAction()`) are there simply to handle regular HTTP requests. Even the JSON-RPC requests don't go through them. It is there to implement the traditional HTTP/HTML functionality by explicitly calling the

mapper. The `jsonrpcAction()` method is simply a placeholder to declare the URL where a user would go to use the JSON-RPC interface, which requires a starting HTML page where the JavaScript is loaded and run off of.

Before we move on to actually accessing the data we'd like to bring up another point. Because we have built in support for 4 different access methods (HTTP, HTML w/ JavaScript, AMF and XML-RPC) testing in the traditional sense would be a bit of a bear. But that's where one of the real benefits of using the Data Mapper design pattern lies. Because the business logic is completely distinct from the service calling it we can test every single piece of functionality without any HTTP calls. That means that, besides implementing the front end, you should have identical results, automatically, for all 4 request types. For the sake of brevity we will not demonstrate the unit testing, though the tests are available in the downloadable workspace.

Flex your mobile muscle

On the Flash side, I've created a basic Flex application. While Flex can be heavier than creating an application that doesn't use the Flex Framework, on a lot of devices we've found that Flex performs fairly well. In a lot of cases, if you're building a multiscreen application, you'll want to design for mobile first. Because the mobile form factor has so many unique properties it can be easier to scale up than trying to turn an application into a mobile one after the fact.

I did a few things to this application to make it more mobile friendly. By default, the mapping API in Flash doesn't support multi-touch and the default controls are too small to use on the smaller screen. So I created big buttons for zooming in and out. I tried to make them transparent so that they didn't interfere with the map as much. I also kept transitions to an absolute minimum. In this case the slide left/right seems to be what most mobile applications are using so I stuck to that.

Diving into the code there's not much that's strange. Since the magic happens on the server side, we can just code the Flex application as normal. Start with the components for the Map and the buttons for zooming in/out see (Listing 3).

Listing 3

```
<maps:Map id="map" key="<Your google Maps API Key Here>" width="100%" height="100%"
    doubleClick="map_doubleClickHandler(event)" mapevent_mapready="map_mapevent_mapreadyHandler(event)"
/>
<s:Button id="zoomIn" alpha=".5" width="100" height="100" x="5" y="5" fontSize="50" label="+"
click="zoomIn_
clickHandler(event)" />
<s:Button id="zoomOut" alpha=".5" width="100" height="100" x="5" y="115" fontSize="50" label="-"
click="zoomOut_clickHandler(event)" />
<components:PeakDetail id="peakDetail" width="480" height="800" x="480"/>
```

I've also got a custom component that deals with showing detailed information about the peak when it is selected. Put that file in the components directory. There are a couple of semi-unique things here. One is that we create an ActionScript class to represent the peak object. Then create a peak variable and make it bindable. That lets us set the variable from the main application and it will change the content in the custom component. The custom component also uses the `FlexGlobals.topLevelApplication` class which is new in Flex 4 to refer to the main application see (Listing 4 below).

Listing 4

```
<?xml version="1.0" encoding="utf-8"?>

<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx" width="480" height="800"
  xmlns:styles="com.google.maps.styles.*"
">

  <fx:Script>

    <![CDATA[
      import mx.core.FlexGlobals;
      import spark.components.Application;
      import vo.Peak;
      [Bindable]
      public var peak:Peak;
      protected function btnBack_clickHandler(event:MouseEvent):void
      {
          FlexGlobals.topLevelApplication.moveRight.play();
      }
    ]]>
  </fx:Script>
  <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
  </fx:Declarations>
  <mx:Image source="@Embed('assets/background.png')" width="100%" height="100%" alpha=".5" />
  <s:Button id="btnBack" x="350" y="5" width="100" height="40" label="Back"
    click="btnBack_clickHandler(event)"/>
  <s:Label x="10" y="100" fontSize="25" text="Name" />
  <s:Label x="10" y="150" fontSize="25" text="Prominence" />
  <s:Label x="10" y="200" fontSize="25" text="State" />
  <s:Label x="10" y="250" fontSize="25" text="Range" />
  <s:Label x="150" y="100" fontSize="25" text="{peak.name}" />
  <s:Label x="150" y="150" fontSize="25" text="{peak.prom}" />
  <s:Label x="150" y="200" fontSize="25" text="{peak.state}" />
  <s:Label x="150" y="300" fontSize="25" text="{peak.range}" />
</s:Group>
```

Then in the `fx:Declarations` tag back in the main application file we can add the transitions and the `RemoteObject` and `CallResponder` tags for dealing with the data from the service.

```
<s:RemoteObject id="srv" destination="zendamf" source="Peaks" fault="srv_faultHandler(event)" />
<s:CallResponder id="peakserviceResult" result="peakserviceResult_resultHandler(event)" />
```

The next step is to add the event handlers. Set up event handlers for the zoom in/zoom out buttons and some code that runs when the map is ready see (Listing 5).

Listing 5

```
protected function zoomIn_clickHandler(event:MouseEvent):void
{
    map.zoomIn();
}
protected function zoomOut_clickHandler(event:MouseEvent):void
{
    map.zoomOut();
}
protected function application1_creationCompleteHandler(event:FlexEvent):void
{
    peakserviceResult.token = srv.getMountains('Colorado','Southern Rocky
Mountains');
}
protected function map_mapevent_mapreadyHandler(event:MapEvent):void
{
    map.setMapType(MapType.PHYSICAL_MAP_TYPE);
    map.setDoubleClickMode(MapAction.ACTION_NOTHING);
    map.setCenter(new LatLng(39.000,-105.000),6);
}
```

We also have to handle the remote call. Because most of the magic is happening on the server, we don't have to do anything special. When the application finishes loading, the event handler calls the service and then the event handler on the CallResponder loops through the received data and plots it on the map. Then each marker has a click handler so the user can select it and get more detail on it see (Listing 6).

Listing 6.

```
protected function peakserviceResult_resultHandler(event:ResultEvent):void
{
    var len:int = event.result.length;
    for(var i:int=0;i<len;i++)
    {
        var peak:Peak = new Peak();
        peak.name = event.result[i].name;
        peak.prom = event.result[i].prom;
        peak.range = event.result[i].range;
        peak.state = event.result[i].state;

        var marker:PeakMarker = new PeakMarker(new
LatLng(event.result[i].latitude,event.result[i].longitu
de),peak,null);
        marker.addEventListener(MapMouseEvent.CLICK,onClick);
        map.addOverlay(marker);
    }
}
private function onClick(event:MapMouseEvent):void
{
    peakDetail.peak = event.target.peak;
    moveLeft.play();
}
protected function srv_faultHandler(event:FaultEvent):void
{
    trace('problem with getting data.');
```

There are a couple of other classes that have been added to make it easier to pass data around. I've extended the Marker class from Google Maps to create my own PeakMarker which takes a peak object so that I can reference it in the event. The PeakMarker code is very straight forward and the value object just contains all of the properties for the peak. You can find the two classes in the example code.

The last step is to make sure that your application is using a correct services-config.xml file. That is the file that points our Flex application to the PHP code so we can call the PHP methods directly. The services-config.xml file can be found below and you use it by modifying the Flex Compiler settings in the Flex Project properties panel. You can add an Additional compiler argument that says `-services location of your services-config.xml file`. You'll also have to modify the services-config file below to make sure it

points to the correct gateway on your server see (Listing 7).

Listing 7.

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
  <services>
    <service id="amfphp-flashremoting-service" class=
      "flex.messaging.services.RemotingService"
      messageType="flex.messaging.messages.RemotingMessage">
      <destination id="zendamf">
        <channels>
          <channel ref="zend-amf-channel"/>
        </channels>
        <properties>
          <source>*</source>
        </properties>
      </destination>
    </service>
  </services>
  <channels>
    <channel-definition id="zend-amf-channel" class="mx.messaging.channels.AMFChannel">
      <endpoint uri="http://localhost/Contexts/public/"
        class="flex.messaging.endpoints.AMFEndpoint"/>
    </channel-definition>
  </channels>
</services-config>
```

Conclusion

What we have highlighted in this article is how you can easily reproduce functionality in several different mobile formats. What we have not done is demonstrate the full functionality, since that requires a lot of code examples. However, that full functionality is available in the downloadable workspace. So, when you want to build an application that can easily switch between browsers, mobile, JavaScript, AMF, XML there are a few things to remember.

- Do not put business logic inline. Put it in defined blocks, preferably PHP classes, and call it from your entry points. This automatically makes it more service-friendly and also makes it much more testable.
- Don't implement your own service layer. There are so many out there with the Zend Framework examples as one of many.

KEVIN SCHROEDER, RYAN STEWART