Web Service Faceplates

Stuart Celarier, Fern Creek Corporation
stuart@ferncrk.com

Draft 0.41
30 September 2002

This is an early draft of a work in progress, distributed for review purposes. A future version will be submitted for publication. I invite your feedback, comments and criticisms. Do not distribute without express permission.

The code download for this paper is at http://www.ferncrk.com/faceplates.

Introducing Web Service Faceplates

A web service faceplate is a client for a web service that provides a lightweight user interface for the web service.

- HTML is the universal user interface language. DHTML is an imperfect vehicle for manipulating the HTML DOM; current lack of portability.
- Faceplates are modular: can be aggregated in a portal-style HTML page
- Generally focused on a small number of web service calls
- XSLT goodness
    - Transformational grammars are cool, XSLT is an elegant tool
    - Craig Andera's talk at the 2002 Web Services DevCon West on implementing a web service using XSLT filled in the server side of the story
    - Web Service Faceplates provide a compelling story for using XSLT on the client side
- Maximal declarative programming with XSLT.
- MVC architecture supports sophisticated user interface while minimizing calls to the web service by maintaining state

A brief history of faceplates. The faceplate concept originated with embedded controllers, as physical device providing access to tiny user interface for monitoring or changing the controller. As embedded controllers became accessible across a network, physical faceplates were replaced by virtual faceplates on a computer screen. Web Service Faceplates provide the same style of lightweight user interfaces for a web service.

The concept of web service faceplates are introduced in [Mohr]. My specific contributions are in the following areas:

- Improved initialization and startup.
- Improved data model.
- Better code organization for clarity of exposition and to assist the developer.

- Removed dependency on MSXML 2.0, although it still requires both MSXML 3.0 and 4.0, I am still working to reduce that to MSXML 4.0 alone.
- Completely new example implemented from the ground up to better illustrate the web service faceplate concept in a single application.

Reference client

Among the attributes of web service faceplates, several of them commend faceplates to being used as a reference client

- Easy to write: a faceplate application is an XML file that contains XSLT (for declarative programming) and JavaScript (for procedural programming), plus a tiny auxiliary XSLT file used to bootstrap the application into existence as DHTML
- Transparent: as XML, a developer of a web service client can use a faceplate application as guidance on creating
- Easy to validate: see "Further directions" for an obvious next step, which is validating the faceplate application XML against an XSD schema. This gives rise to the powerful concept of schema-based programming.
- Appropriate for testing implementation and availability of the web service
- Make no use of WSDL. One or more faceplate applications could be used as the reference implementation to specify and test the Web Service
- Theoretically portable because of its dependence on open standards like XML, XSLT and HTML. As the integration of XML into HTML improves, there is good reason to believe that current barriers will disappear. Current barriers to cross-platform portability are:
    - Inconsistent DHTML implementations: see "Futher directions" for using SVG for a consistent user interface substrate
    - No W3C specification on invoking an XSLT processor from HTML.

Faceplate Design Objectives

Web service faceplates should be modular and lightweight, so they are suitable for use in portal application. Modularity is expressed as an HTML page that can be embedded inline in HTML using an `<iframe>` element. Lightweight here means sufficiently conservative with resources so as to coexist with other lightweight modules in a parent HTML page. This includes minimizing round trips to the web service.

Maximal use of XSLT, declarative programming is generally easier to support. Errors and exceptional conditions lead to empty or incorrect results, not crashes.

Support the MVC pattern in the architecture so that faceplates can have sophisticated user interfaces. To support the MVC pattern, we need an event mechanism so that views can be redrawn when the corresponding data in the model is changed.

A layered architecture in the procedural code eases development process.

A Sample Weather Web Service

It is difficult to talk for long about developing clients of web services without having a web service to talk about. I've implemented a sample web service as an ASP.NET application in C# which provides weather information for select locations around the globe. Code for this web service and the web service faceplate are available from the code download site listed at the top of this paper.

The Weather web service exposes three methods:

- `GetCountries()` returns a list of countries, where each country is composed of a name and abbreviation pair;
- `GetStations( countryAbbr )` returns a list of weather stations for the specified country, where each station is also composed of a name and abbreviation pair; and
- `GetWeatherReport( stationAbbr )` returns a name, temperature and forecast for the specified weather station.

Web Service Faceplate for the Weather Web Service

The aim of creating a faceplate for the Weather web service is to provide enough user interface for the faceplate to be useful and valuable without having to devote a lot of resources to development or maintenance.

Figure 1 illustrates the intended use of a web service faceplate as a modular, encapsulated application, suitable for aggregated use in a portal-style application. The web service faceplate is an XML file which contains procedural code (JavaScript), declarative code (XSLT), and data. Using the web service faceplate is accomplished by naming the XML file as the source of an `<iframe>` element in HTML. Optionally, the `<iframe>` element can be styled to make its appearance consistent with the hosting HTML page. **Figure 2** shows the web service faceplate in action.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
  <head>
    <title>Test Container for Weather Web Service Faceplate</title>
    <style type="text/css">
      body { background-color: gray }
      iframe.mvc {    TOP: 50px;   LEFT: 50px;
                    HEIGHT: 350px; WIDTH: 500px;
                  POSITION: absolute }
    </style>
  </head>
  <body>
    <h1>MVC Faceplate</h1>
    <iframe class="mvc" src="http://localhost/MvcFaceplate/MvcFaceplate.xml"
            scrolling="no" frameborder="0"></iframe>
  </body>
</html>
```

Figure 1: Sample use of web service faceplate

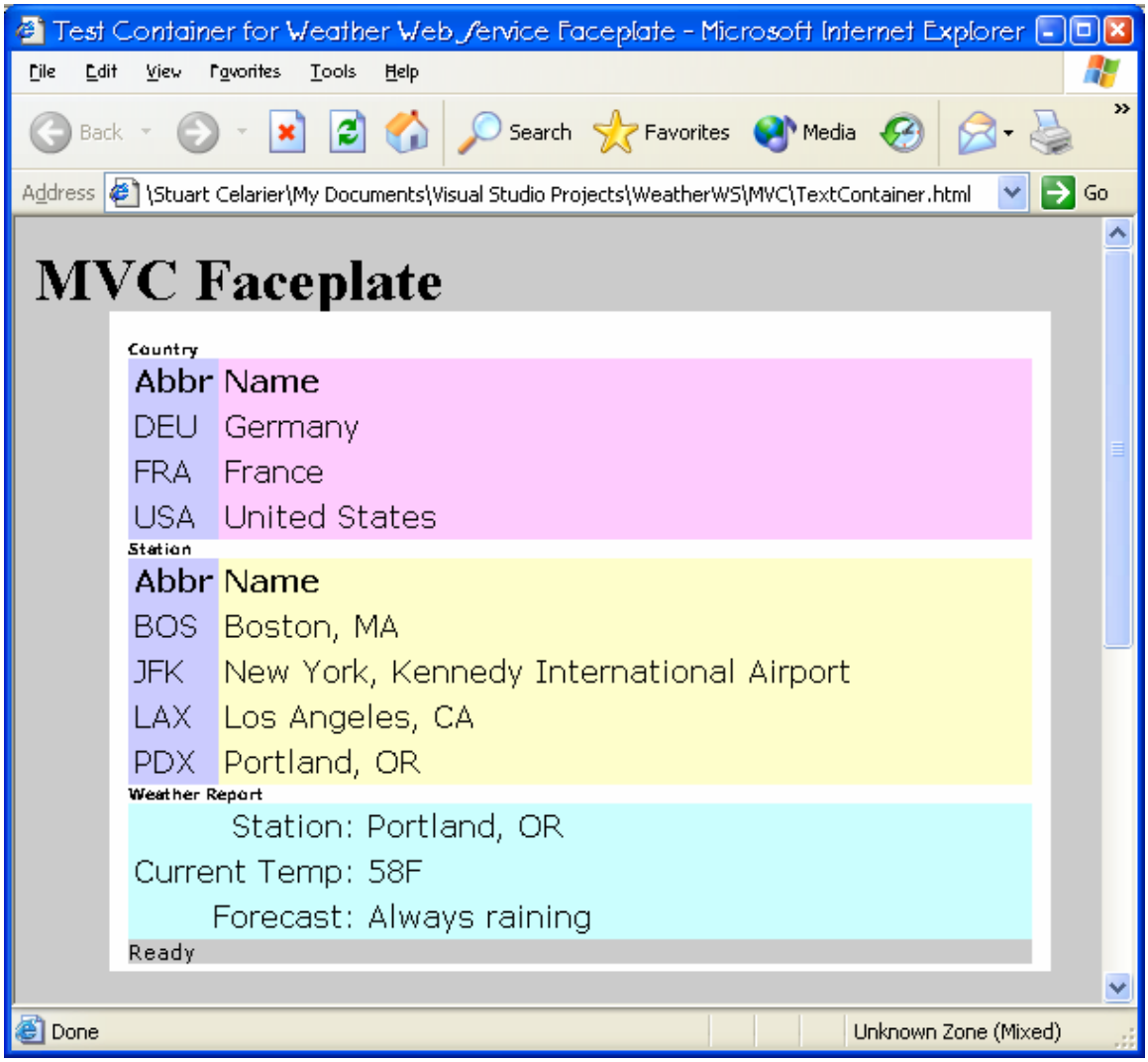

Figure 2: The web service faceplate in action

Faceplate Anatomy

Figure 3 shows the overall structure of the web service faceplate application, which is an XML file. First, notice the use of an `<?xml-stylesheet?>` processing instruction which will invoke an XSLT transformation to turn this faceplate application XML file into HTML That initialization transformation will be examined shortly.

The `<Faceplate>` document element is composed of several different child elements. The `<Model>` element contains initial data for the data model in the MVC architecture. For this simple application the initial data model is empty, but it can be used to hold data to display while the first web service calls are being made or in case the web service is not available.

The `<Views>` element contains multiple nested `<View>` elements, providing a flexible set of views in the MVC architecture. The `<View>` elements will be transformed into HTML `<DIV>` elements with unique IDs by the initialization transformation.

Accompanying the multiple views, the `<Style>` element contains CSS properties for styling the views.

The `<CodeBlock>` element contains all the procedural code, i.e., JavaScript, for the faceplate application. The procedural code has been partitioned into three layers: application specific, application general, and system services. These layers organize the code and help gauge how much of the code is custom for this specific application, how much is custom for this general style of application, how much is supporting common system services. The `<Layer>` elements are solely for the benefit of the faceplate application developer, the initialization transformation will place all their text contents into a single HTML `<SCRIPT>` element.

The `<Events>` element contains multiple `<Event>` elements which are used by an event dispatch mechanism in the procedural code. Events constitute some declarative high-level glue used to connect the procedural and declarative code together in the faceplate application.

Lastly, the `<Intentions>` element contains all the declarative code, i.e., XSLT documents, for the faceplate application. The term ‘intention’ is fitting, since we are using a declarative language to *declare our intentions*. The declarative code has been partitioned into four regions by the type of transformation. The `Startup` intention type contains XSLT transforms used at application startup time, after the initialization transform has converted this document to HTML. The `SoapRequest` intention type contains transforms that produce SOAP request messages for the different web service methods used by the faceplate application. The `UpdateModel` intention type contains transforms that modify the data model. There are two sources for updating the data model: user interactions and data returned in SOAP response messages. The `View` intention type contains transforms that render individual views based on the data model.

Taken together, the `<CodeBlock>`, `<Events>` and `<Intentions>` elements form the controller in the MVC architecture.

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/xsl" href="MvcFaceplateInit.xslt"?>

<Faceplate>
  <Model/>

  <Views>
    ...
  </Views>

  <Style>
    ...
  </Style>

  <CodeBlock>
    <Layer id="AppSpecific" >...</Layer>
    <Layer id="AppGeneral"  >...</Layer>
    <Layer id="System"      >...</Layer>
  </CodeBlock>

  <Events>
    ...
```

```
  </Events>

  <Intentions>
    <IntentionType id="Startup"     >...</IntentionType>
    <IntentionType id="SoapRequest" >...</IntentionType>
    <IntentionType id="UpdateModel" >...</IntentionType>
    <IntentionType id="View"        >...</IntentionType>
  </Intentions>
</Faceplate>
```

Figure 3: High-level structure of the web service faceplate

Faceplate Initialization

Let's turn our attention to the initialization transformation shown in **Figure 4**. The aim here is to do the bare minimum in this transformation, preferring to include startup processing in the faceplate application XML document instead. The DHTML DOM provides read-only access to `<script>` and `<style>` elements, so they must be initialized before the DOM is created. In both cases, they use the default XSLT templates to concatenate all the text node descendants of the `<CodeBlock>` and `<Style>` elements, respectively, in the faceplate application XML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <script>
          <xsl:apply-templates select="/Faceplate/CodeBlock"/>
        </script>
        <style>
          <xsl:apply-templates select="/Faceplate/Style"/>
        </style>
      </head>
      <body id="body">
      </body>
    </html>
  </xsl:template>

</xsl:transform>
```

Figure 4: Initialization XSLT

Continuing on to the rest of the startup of the faceplate application, in **Figure 5** we see the `Initialize` function being bound to the `onload` method of `window`, so that function will be executed at startup. Two global variables are also initialized: `doc` refers to the underlying faceplate application XML document, and `model` is an XML DOM object initialized with the `<Model>` element in the faceplate application XML.

The `Initialize` function performs an XSLT transformation (via the `TransformObject` function) using the faceplate application XML document as the source and the XSLT from the intention with ID of `Bootstrap`.

```
  <CodeBlock>
    <Layer id="AppSpecific">
      window.onload = Initialize;

      var doc = document.XMLDocument;
```

```
      var model = CreateDOMDocument(
                              doc.selectSingleNode("/Faceplate/Model").xml );

      function Initialize()
      {
        body.innerHTML = TransformObject( doc,
                                          GetIntentionTransform("Bootstrap") );

        // bind Update method to views
        CountriesView.Update      = UpdateView;
        StationsView.Update       = UpdateView;
        StatusView.Update         = UpdateView;
        WeatherReportView.Update  = UpdateView;

        Notify( "EventGetCountries" );
      }
      ...
    </Layer>
    ...
  </CodeBlock>
```

Figure 5: Initialize function

Let's drill down into the `Bootstrap` intention in **Figure 6**. Each `<Intention>` element contains an XSLT document, with the XSLT namespace declared on the parent `<Intentions>` element. Following the convention used in [Mohr], we prefer the `<xsl:transform>` as a more appropriate XSLT document element name than its synonym `<xsl:stylesheet>`.

The source document for the `Bootstrap` intention is the faceplate application XML, and we see that this XSLT is transforming multiple nested `<View>` elements into HTML `<div>` elements with `id` and `class` attributes so that they can be addressed and styled, respectively. The value of the `id` and `class` attributes is the concatenation of the original `<View>`'s `id` value and the string "View". The third template in this transform is simply the identity transform in XSLT, copying all nodes and attributes into the result.

```
<Intentions xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <IntentionType id="Startup">

    <Intention id="Bootstrap">
      <xsl:transform version="1.0">
        <xsl:output method="html"/>

        <xsl:template match="/">
          <xsl:apply-templates select="/Faceplate/Views/View"/>
        </xsl:template>

        <xsl:template match="View">
          <div id="{concat(@id, 'View')}" class="{concat(@id, 'View')}">
            <xsl:apply-templates select="node()"/>
          </div>
        </xsl:template>

        <xsl:template match="node()|@*">
          <xsl:copy>
            <xsl:apply-templates select="node()|@*"/>
          </xsl:copy>
        </xsl:template>

      </xsl:transform>
    </Intention>
```

```
    </IntentionType>
    ...
  </Intentions>
```

Figure 6: The Bootstrap intention

Returning to the `Initialize` function of **Figure 5**, the result of this transformation is set as the `innerHTML` in the `<body>` element. This is precisely what we want: the faceplate application is rendered as a set of nested views expressed as HTML `<div>` elements that can be addressed and styled independently.

Next in the `Initialize` function we find some DHTML code, defining an `Update` method on various views and binding the `UpdateView` function to that method. The `UpdateView` function, **Figure 7**, is in the application general layer of the `<CodeBlock>`, and replaces the object's contents with the result of an XSLT transformation that uses the data model as its source.

```
function UpdateView( intentionTransform )
{
  try
  {
    this.innerHTML = TransformObject( model, intentionTransform );
  }
  catch(e)
  {
    alert("Error during transform:\n\n"+e.description);
  }
}
```

Figure 7: UpdateView function

Lastly, in the `Initialize` function the faceplate application is started with the call to the `Notify` function with an event that will result in the first web service call being made and the user interface being updated accordingly. **Figure 8** shows the application specific `Notify` function, and **Figure 9** shows the `<Events>` element from the faceplate application XML.

The `Notify` function gets an event ID as input, finds the `<Event>` element with that ID, and parses the space-delimited string of notifications that need processing. For each notification it determines if it is associated with a specific procedural call (the case statements), in which case the corresponding call is invoked. Otherwise the notification is taken to be the ID of a declarative intention. Such intentions have a `boundTo` attribute, specifying the view that that intention updates.

```
function Notify( event )
{
  var notify = GetEvent( event ).getAttribute("notify").split(" ");
  for ( each in notify )
  {
    switch( notify[each] )
    {
      case "GetCountries"     : GetCountries();     break;
      case "GetStations"      : GetStations();      break;
      case "GetWeatherReport" : GetWeatherReport(); break;

      default:
        var intention = GetIntention( notify[each] );
```

```
        document.all( intention.getAttribute("boundTo") ).Update(
                                                    GetTransform( intention ) );
    }
  }
}
```

Figure 8: Notify function

```
<Events>
  <Event id="EventGetCountries" notify="StatusBusy GetCountries
                                        ViewCountriesByAbbr StatusReady"/>
  <Event id="EventGetStations" notify="StatusBusy GetStations
                                       ViewStationsByAbbr StatusReady"/>
  <Event id="EventGetWeatherReport" notify="StatusBusy GetWeatherReport
                                            ViewWeatherReport StatusReady"/>
  <Event id="EventViewCountriesByAbbr" notify="ViewCountriesByAbbr"/>
  <Event id="EventViewCountriesByName" notify="ViewCountriesByName"/>
  <Event id="EventViewStationsByAbbr" notify="ViewStationsByAbbr"/>
  <Event id="EventViewStationsByName" notify="ViewStationsByName"/>
</Events>
```

Figure 9: Event declarations

The call to `Notify` that appears in the `Initialize` function passed the `EventGetCountries` ID. Locating the corresponding `<Event>` element in Figure 9, we see the notifications are `StatusBusy`, `GetCountries`, `ViewCountriesByAbbr` and `StatusReady`. This specifies a sequence of processing: update the Status view to read “Busy”; perform a SOAP call to get the list of countries and update the model accordingly; have the Countries view update itself from the model; and update the Status view to read “Ready”. Upon completing the initialization, the faceplate looks like that shown in **Figure 10**.

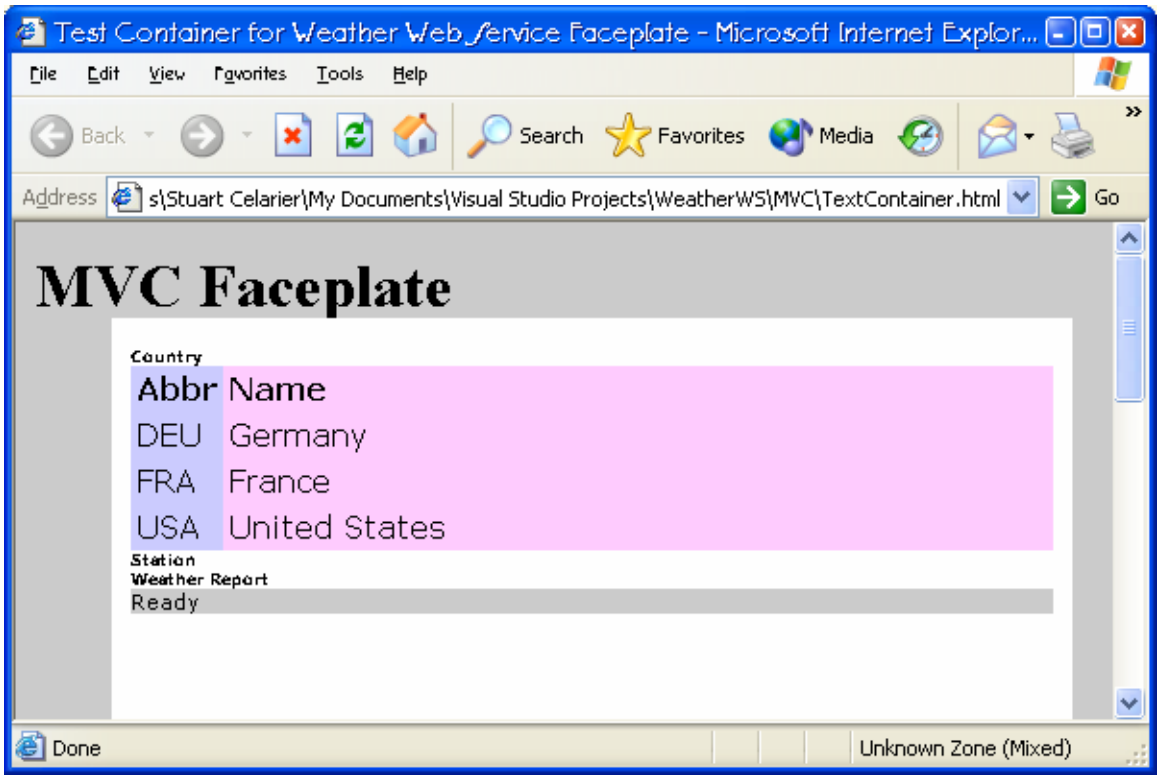

Figure 10: The initialized faceplate

Making SOAP Calls

From the `Notify` function we saw that there are a few notifications that correspond to procedural code to make SOAP calls. **Figure 11** shows the `GetCountries` function as an example, along with the helper function `WeatherWSSender`. The helper function takes the name of an action and returns an XMLHTTP object with all the headers set to post a SOAP request to the Weather web service.

```
function GetCountries()
{
  var sender = WeatherWSSender( "GetCountries" );
  sender.send( TransformObject( doc,
                          GetIntentionTransform("SoapRequestGetCountries") ) );

  if ( sender.status == 500 )   //  SOAP fault
    alert( sender.responseText );
  else
  {
    if ( sender.status == 200 )
      model.loadXML( TransformObjectParams( sender.responseXML,
                              GetIntentionTransform("UpdateModelGetCountries"),
                              CreateParam( "model", model ) ) );
  }
}

...

function WeatherWSSender( action )
{
  var sender = new ActiveXObject("Msxml2.XMLHTTP.4.0");
  sender.open("POST","/WeatherWS/Weather.asmx",false);
  sender.setRequestHeader("Content-Type","text/xml");
  sender.setRequestHeader("charset","UTF-8");
  sender.setRequestHeader("Host","localhost");
  sender.setRequestHeader("SOAPAction",
                          "urn:ferncrk.com:webservices:Weather/" + action );
  return sender;
}
```

Figure 11: Making a SOAP call

`GetCountries` then uses the XMLHTTP object to send (post) the SOAP request generated by the intention `SoapRequestGetCountries` which is shown in **Figure 12**. This is a very simple SOAP request message to produce, since the corresponding web service method takes no parameters. The other two SOAP request intentions (not shown here) do illustrate passing parameters into the web service method call.

```
<Intention id="SoapRequestGetCountries">
  <xsl:transform version="1.0">

    <xsl:template match="/">
      <soap:Envelope
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
        <soap:Body>
          <GetCountries xmlns="urn:ferncrk.com:webservices:Weather"/>
        </soap:Body>
      </soap:Envelope>
    </xsl:template>
```

```
  </xsl:transform>
</Intention>
```

Figure 12: Creating a SOAP request message

The XMLHTTP performs the post synchronously, and on return checks for failure. If the SOAP call succeeded, a second XSLT transform is used to extract data from the response and update the model with that data. Actually, that is what we'd like to do, but in reality we presently have to create an entirely new copy of the data model, serialize it as XML, then reparse it in to the model using `loadXML`. Although inelegant and inefficient, so far it appears the only reasonable[1] way of updating the model with the current state of integration of XML in Internet Explorer.

As a consequence, the intentions used to update the model are based on the identity transform to copy everything we are not concerned with, and a few other templates to add or remove the relevant elements to the model. **Figure 13** shows the `UpdateModelGetCountries` intention. The transformation is applied to the SOAP response message (`sender.responseXML`), and the current model is passed in as a parameter to the transformation. The template that matches the root creates a new `<Model>` element, adds a `<Countries>` element inside of that and populates it with the contents of the SOAP body. It also copies the rest of the current model, except for any previous `<Countries>` element. The second template reaches down into the SOAP body to get the interesting data, creating one `<Country>` element in the model for each country returned in the country list.

```
<Intention id="UpdateModelGetCountries">
  <xsl:transform version="1.0"
      xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:weather="urn:ferncrk.com:webservices:Weather"
      exclude-result-prefixes="soap weather">
    <xsl:output omit-xml-declaration="yes"/>
    <xsl:param name="model"/>

    <xsl:template match="/">
      <Model>
        <Countries>
          <xsl:apply-templates select="/soap:Envelope/soap:Body/*"/>
        </Countries>
        <xsl:apply-templates select="$model/Model/*" mode="copyModel"/>
      </Model>
    </xsl:template>

    <xsl:template match="weather:GetCountriesResponse/
              weather:GetCountriesResult/weather:countryList/weather:Country">
      <Country Name="{weather:name}" Abbr="{weather:abbr}"/>
    </xsl:template>

    <xsl:template match="node()|@*" mode="copyModel">
      <xsl:copy>
        <xsl:apply-templates select="node()|@*" mode="copyModel"/>
      </xsl:copy>
    </xsl:template>

    <xsl:template match="Countries" mode="copyModel"/>

  </xsl:transform>
```

[1] As opposed to the MSXML 2.0 hack used in [Mohr]

```
</Intention>
```

Figure 13: Updating the model from a SOAP response message

That completes handling the SOAP request and response, so now it is time to see how the views are updated and the value that caching data in a model provides to web service faceplates.

Updating the View

Figure 14 shows the `ViewCountriesByAbbr` intention. You will recall from looking at the `Notify` function that intentions which update view contain a `boundTo` attribute specifying a binding to a view. Naturally enough, the `ViewCountriesByAbbr` intention is bound to the `CountriesView` view. This intention generates a table with a header row and one row for each `<Country>` in the model, with the countries sorted by the `Abbr` attribute. The table data item, `<td>`, for the Name column adds an `onclick` attribute that uses the `Notify` function to redraw the view sorted by name. Thus redrawing views is accomplished using data in the model, and does not involve unnecessary calls to the web service.

```
<Intention id="ViewCountriesByAbbr" boundTo="CountriesView">
  <xsl:transform version="1.0">
    <xsl:output method="html"/>

    <xsl:template match="/">
      <table border="0" cellpadding="3" cellspacing="0">
        <tr class="headerRow">
          <td class="sortColumn">Abbr</td>
          <td onclick="Notify('EventViewCountriesByName')">Name</td>
        </tr>
        <xsl:apply-templates select="/Model/Countries/Country">
          <xsl:sort select="@Abbr"/>
        </xsl:apply-templates>
      </table>
    </xsl:template>

    <xsl:template match="Country">
      <tr>
        <xsl:attribute name="onclick">
          SelectCountry("<xsl:value-of select="@Abbr"/>")
        </xsl:attribute>
        <td class="sortColumn">
          <xsl:value-of select="@Abbr"/>
        </td>
        <td>
          <xsl:value-of select="@Name"/>
        </td>
      </tr>
    </xsl:template>

  </xsl:transform>
</Intention>
```

Figure 14: ViewCountriesByAbbr intention

The table row for each country has an `onclick` attribute that uses the `SelectCountry` function, **Figure 15**, to select a country. This simply creates an XML DOM object containing the selected country, updates the model with the selected country, and fires off

an event to get the station information for the selected country through another web service method call..

```
function SelectCountry( country )
{
<![CDATA[
  var source = CreateDOMDocument( "<SelectedCountry>" + country +
                                  "</SelectedCountry>" );
]]>
  model.loadXML( TransformObjectParams( source,
                           GetIntentionTransform("UpdateModelSelectedCountry"),
                           CreateParam( "model", model ) ) );
  Notify( "EventGetStations" );
}
```

Figure 15: SelectCountry function

This completes the tour through the major functionality of the web service faceplate. All the code that remains unexamined is boilerplate copies of the code we've looked at, and some low level support.

Further directions

The basic concepts of web services faceplates were introduced in [Mohr]. They proceed from this point to develop the following ideas.

- An XML schema document can be constructed in parallel with framework application development
  - Use to validate framework applications
  - Development can focus on the framework application schema
  - Schema-Based Programming
- JSML, JavaScript Markup Language, www.jsml.com
- Model-Based Programming
- Simulating applications with Petri nets

Another direction which I find merits particular attention is using Scalable Vector Graphics (SVG) for the presentation of views in web services faceplates. SVG offers the following benefits:

- SVG DOM has a uniform implementation cross platform, DHTML is lousy and lossy for cross platform deployment
- Painter model much richer than DHTML
- Richer user interface elements: animation, alpha channel, better events, compatible with SMIL

References

[Mohr] Stephen Mohr, Michael Corning, Erik Fuller, Michael John and Donald Kackman, *Web Service Faceplates: Building Web Service Clients using XML and XSLT*. Wrox Press, 2002.

[JSML] JavaScript Markup Language website: http://www.jsml.com