CHAPTER 4

AJAX AND REST

4.1 INTRODUCTION

AJAX is a name applied to a set of programmatic techniques that enable browsers to communicate *asynchronously* with web server. Common uses of AJAX include retrieving content from the server to be inserted into the current page and transmitting new or update information to be persisted on the server. AJAX techniques make it possible to achieve these results without causing a total refresh or re-rendering of the current page.

Ajax incorporates several pre-existing technologies such as:

- standards-based presentation using XHTML and CSS;
- dynamic display and interaction using the Document Object Model;
- data interchange and manipulation using XML and XSLT;
- asynchronous data retrieval using XMLHttpRequest;
- and JavaScript binding everything together.

AJAX stands for either ***A**synchronous **J**avascript **A**nd **X**ML or **A**synchronous **J**avascript **A**nd **X**MLHttpRequest*. AJAX does not necessarily make use of XML but it almost always involves both Javascript and the XMLHttpRequest object. Just as DHTML can be thought of as "*Javascript, CSS and HTML DOM*", AJAX can be summarized as "*DHTML and XMLHttpRequest (XHR)*".

Microsoft originally released the XHR object in 1999 with Windows IE 5 as an ActiveX object available through the use of Javascript and VBScript. It is now supported by FireFox, Chrome, Safari, Opera by using a native Javascript object. Although the technologies have been in existence and used by some developers in the past, it has only recently gained large popularity, also based on the support offered by browsers.

4.2 AJAX WITH HTML HIDDEN FRAME

Before the creation of the XMLHttpRequest object by Microsoft, HTML frames were used as a vehicle for submitting background request and accepting responses.

This technique is based on the use of a main frame, a secondary frame and a JavaScript code.

The main frame is responsible for the interaction with the user and for the presentation of information.

The secondary hidden frame is used for background request and response.

The JavaScript code in the main content frame passes information to the JavaScript code in the hidden frame, which submits an HTTP request. The response to this request refreshes the hidden frame, triggering additional JavaScript code that passes information and control back to the main content frame.

Let’s see some example just for understanding the mechanism.

<u>Example with GET request</u>

The web page composed by more than one frame is built using the tag <frameset>, which includes the tag <frame> for each frame to visualize[20]. Let’s see the various web pages involved in order to do a GET request in an asynchronous way.

```
<html>
      <head>Ajax using hidden frame</head>
      <frameset rows=”100%, 0” style=”border:0”>
       <frame name=”mainframe”   src=”main.html”
      noresize=”noresize” />
```

[20] The tags <frameset> and <frame> are not longer supported in the HTML 5.

```
      <frame name="hiddenframe" src="about:blank"
     noresize="noresize" />
     </frameset>
</html>
```

The attribute rows of *<frameset>* element contains the dimension of each frame separated by a comma. In the previous example the visualization frame will have all the space, while the hidden frame will be high zero pixel.
Note the attribute *noresize* to prevent the user from scaling up the frames. In this way the browsers will block the user from seeing inside the communication frame. In the visualization frame we have linked the file main.html while in the communication frame there is nothing, that is about:blank. Here is the file main.html.

```
<html>
     <body>
          <script GetAsynData () {
            top.frames['hiddenFrame'].location="data.html";
            }
          </script>
     <form>
          <input name="confirm" type="button"
          Value="Get Data" onclick="GetAsynData();" />
     </form>
     </body>
</html>
```

The JavaScript function uses the top object of the browser window to assign to the location property of hidden frame the web page to request to the web server. When the browser locates the value of location property, it updates the frameset loading the page data.html, which has the following content.

```
<html>
      <body>
            <script>
                  Window.alert(“Data received!”);
            </script>
      </body>
</html>
```

When the user clicks on the button Get Data the message “Data received!” appeared on a dialog box.

Example with POST request

If we use a POST request the structure of the involved web pages is a little bit different. We are going to implement a server functionality which inverts a string using PHP language on the web server. The frameset structure is equal to the example with the GET request.

```
<html>
      <head>Ajax using hidden frame</head>
      <frameset rows=”100%, 0” style=”border:0”>
      <frame name=”mainframe”   src=”form.html”
      noresize=”noresize” />
      <frame name=”hiddenframe” src=”about:blank”
      noresize=”noresize” />
      </frameset>
</html>
```

The content of form.html is:

```
<html>
      <body>
      <fom action =”reverse.php” target=”hiddenFrame”
      method=”POST” />
      <input name=”name” length=”30” /><br />
      <input name=”confirm” type=”submit” value=”Get in reverse
      mode” />
      </form>
      </body>
</html>
```

In the tag *<form>* the target attribute contains the name of the hidden frame to use as target. The data arrives to the web page reverse.php which contains the server side logic of web application. Here is the content.

```
<html>
      <body>
        <script>
        top.frames['mainFrame'].document.forms[0].name.value =
        "<?php =strrev($_POST['name']); ?>";
        </script>
      </body>
</html>
```

Advantages in using hidden frames

The main benefit in using hidden frames in Ajax applications is the browser preservation of navigation history. The user can use the back and the forward buttons as it was a normal web application or an ordinary web site. This is very important for the web usability.

Disadvantages in using hidden frames

The main limit in using hidden frames is the impossibility to know what happened to the HTTP request. The frame which manages the communication with the server is unable to get information on the stage reached by the request processing. The web application could be "frozen" in a waiting state forever.

4.3 AJAX WITH HTML INTERNAL FRAME

With HTML 4.0 was introduced a new tag *<iframe>*[21] which stays for internal frame. This tag gives the possibility to insert a frame in a HTML page without the necessity to define a frameset.

Example with GET request

The internal frames are managed in the same way as the hidden ones. Here is an example.

```
<html>
      <body>
      <script>
       Function GetAsynData() {
            top.frames[‘internalFrame’].location = “data.html”
      }
      </script>

      <form>
        <input name=”confirm” type=”button”
        value=”Get Data” onclick=”GetAsynData();” />

        <iframe src=”about:blank” name=”internalFrame”
        style=”display: none”></iframe>

      </form>
</html>
```

The internal frame is present in the web page with a empty content namely we have src=”about:blank”. Moreover the style attribute with *display: none* communicates to the browser not to show the internal frame.

[21] The tag <iframe> is still supported in HTML 5 with new attributes and with other ones no longer supported. It creates an inline frame that contains another document.

Example with POST request

Using a POST request the structure of the web page is a bit different as we can see in the following example.

```
<html>
  <body>
      <form action=”reverse.php” target=”internalFrame”>

        <input name=”name”  length=”30” /><br />
        <input name=”confirm” type=”submit” value=”Get Data” />

        <iframe src=”about:blank” name=”internalFrame”
        style=”display: none”></iframe>

      </form>
  </body>
</html>
```

We have in the calling web page only an internal frame so the reverse.php will be in this way.

```
<html>
      <body>
        <script>
            top.document.forms[0].name.value =
            “<? =strrev($_POST[‘name’]); ?>”;
        </script>
      </body>
</html>
```

4.4 AJAX INTERACTION MODEL

Now we analyze the flow of interaction model from the request to the response. The structure can be summarized in the Fig.4.1.

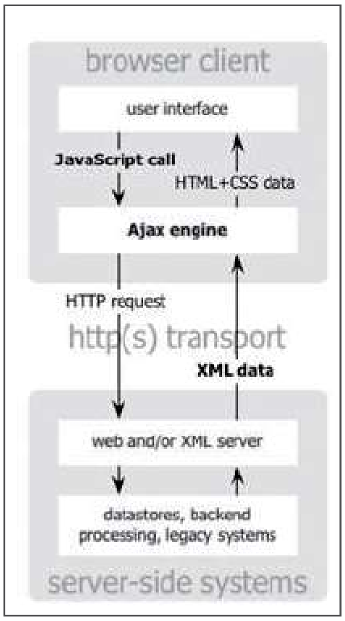

Figure 4.1 – AJAX Interaction Model

<u>AJAX Engine</u>

The XMLHttpRequest (XHR) is the core of the AJAX engine. It is the object that enables a page to GET data from or POST data to the server as a background request, which means that it does not refresh the entire document in the browser window during this process.

This type of interaction model is more intuitive than the standard HTTP request. This is because changes happen on demand when the user makes them, and allow web applications to feel more like desktop applications. The XMLHttpRequest eliminates the need to wait on the server to respond with a new page for each request and allows users to continue to interact with the page while the requests are made in the background.

However, even if the data processing is in the background, the GET and POST methods of the XHR object work the same as standard HTTP request. Using either the POST or the GET method you can make a request for the data from the server and receive a response in any standardized format.

In the Fig.4.2 we can see how a (user) event is managed using the AJAX paradigm together with the abstraction of Model-View-Controller.

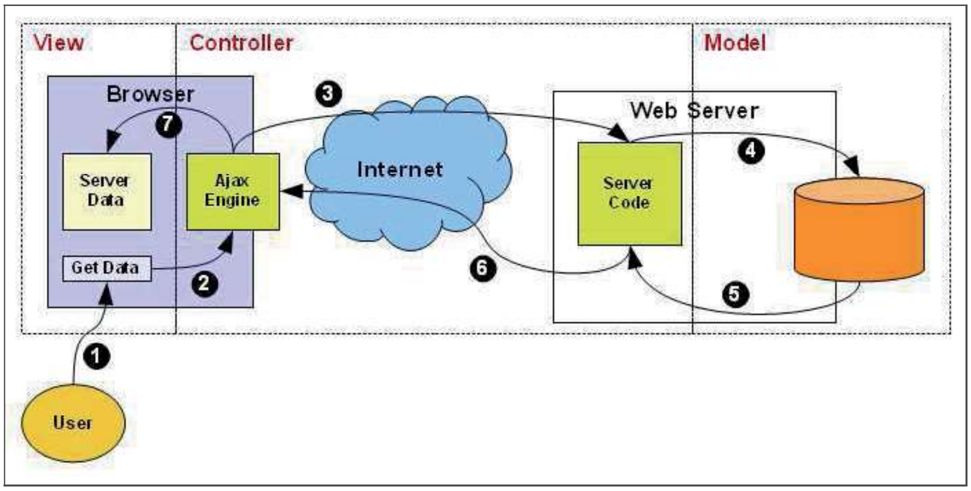

Figure 4.2 – AJAX and MVC event management.

The entire cycle of event is characterized by the following steps:

1) the user click on the button in order to get data;
2) the event is caught by the AJAX engine;
3) the AJAX engine makes a request to the appropriated server service;
4) the server service ask for local resource and services in order to satisfy the request;
5) the local resources and services provide the requested data to the server service;
6) the server service make a response to the AJAX Engine;
7) the AJAX Engine shows the requested data on the browser.

Whereas generally a browser only allows two HTTP persistent connections to a server at anyone time because it trying to be standard compliant to RFC 2616[22], we can make many requests on this two connections. The requests that cannot be immediately managed are parked in an internal queue on the browser. As a consequence a user can make many AJAX requests but they are satisfied with different delays.

Creating the XMLHttpRequest Object and make a Request

All AJAX requests start with a client-side interaction that is typically managed by Javascript. It creates the XHR object and makes an HTTP request to the server.

To create the request object you must check to see if the browser uses the XHR or the Activex object. Windows IE 5 e IE6 use ActiveX object, whereas IE 7 and above, Mozilla FireFox, Opera, Safari and Chrome use the native Javascript XHR object.

```
Function makeRequest(url, callbackMethod)
{
  If (window.XMLHttpRequest)
  {
     XHR = new XMLHttpRequest();
  }
  Else if (window.ActiveXObject)
       {
         XHR = new ActiveXObject("Msxml2.XMLHTTP");
       }
       Else {
         throw new Error("Ajax is not supported by this
browser.");
```

[22] The standard is RFC 2616, "Hypertext Transfer Protocol – HTTP/1.1". Section 8.1.4, covering "*Persistent Connections / Practical Considerations", states: "Clients that use persistent connections SHOULD limit the number of simultaneous connections that they maintain to a given server. A single user client SHOULD NOT maintain more than 2 connections with any server or proxy. A proxy SHOULD use up to 2*N connections to another server or proxy, where N is the number of simultaneously active users. These guidelines are intended to improve HTTP response times and avoid congestion.*"

```
        }
  sendRequest(url, callbackMethod);
}
```

The object can now be used to access all the properties and methods listed in Tables 4.1 e 4.2.

Table 4.1 – XMLHttpRequest Properties

Property	Definition
onreadystatechange	It is fired when the state of request object changes and allows us to set a callback method to be triggered. This property is fired for a total of 4 times.
readyState	Returns number values that indicate the current state of the object. 0 the object is not initialized with data; 1 the object is loading its data; 2 the object has finished loading its data; 3 the user can interact with the object even though it is not fully loaded; 4 the object is completely initialized.
responseText	String version of the response from the server.
responseXML	DOM-compatible document object of the response from the server.
status	Status code of the response from the server.
statusText	A status message returned as a string.

Table 4.2 – XMLHttpRequest Methods

Method	Definition
Abort()	Cancel the current HTTP Request.
getAllResponseHeaders()	Retrieves the values of all the HTTP

	headers.
getResponseHeader("label")	Retrieve the value of a specified HTTP header from the response body.
Open("method","URL"[,asyncFlag[,userName[,"password"]]])	Initializes a request and specifies the method, URL, and authentication information for the request.
Send(content)	Sends an HTTP request to the server and receives a response. It is like clicking the submit button on a form
SetRequest("label","value")	Specifies the value of an HTTP header based on the label.

```
Function sendRequest(url, callbackMethod) {
  XHR.onreadystatechange = function (){
    if (XHR.readyState == 4) {
       if (XHR.status >=200 && XHR.status < 300) {
          callBackMethod;
       };
    }
  };
  XHR.open("GET", url, true);
  XHR.send(null); // GET requests typically have no body
}
```

The *onreadystatechange* is an event handler fired only in asynchronous mode when the state of the request object change and allow us to set a callback method to be triggered. To this property we can assign a reference to a function or build an anonymous function to it as in the above example.

```
  // Assigning a reference to a function
     XHR.onreadystatechange = FunctionName;

  // Building an anonymous function to it
     XHR.onreadystatechange = function() { … };
```

The **open** method of XHR objects takes three parameters. The first is a string that represents the method in which the request is to be sent. This method can be GET, POST or PUT. The second parameter is the URL that is being request in the form of a string, which is XML, JSON, a text file or a server-side language that returns any of these formats. The last parameter is a Boolean value that has a default value of true for asynchronous and false for synchronous.

The **send** method is the actual method that sends the HTTP request and receives a response in the format that you specify. This method takes one string parameter, which can be XML or a simple key/value pair to be sent as a POST.

An AJAX response can come in various formats such as JSON and XML.

XML

XML is composed of custom tags called elements, which are defined in the architecture phase of a web application. They can represent any name, value or data type that will be used in your application. Here is an example:

```
<?xml version=”1.0” encoding=”iso-8850-1” ?>

<categories>
  <category>Priority</category>
  <category>Object<category>
  <category>Expiry Time<category>
  <category>When<category>
  <category>Where<category>
</categories>

<row>
 <items>
      <item> <![CDATA[<u>Hight</u>]]> </item>
      <item> <![CDATA[<b>Project Financial Plan</b>]] > </item>
      <item> 2009-09-06 15:30:00</item>
```

```
     <item    action=”alert('Meeting');”    icon=”img/warn.gif”>
3</item>
     <item> Purple Room </item>
 </items>
</row>

<row>
 <items>
     <item> <![CDATA[<i>Normal</i>]] > </item>
     <item> <![CDATA[<b>Project Management</b>]] > </item>
     <item> 2009-10-12 10:30:00</item>
     <item                            action=”alert('Meeting');”
icon=”img/warn.gif”>2</item>
     <item> White Room </item>
 </items>
</row>

</xml>
```

Let's take a look at attributes and how they help us add additional information to your XML data.

In order to represent an expiry event we have created a group of item that can eventually become a collection of objects when they are parsed on client side.

The item with action attribute means that the action is triggered starting n days before the established meeting time and the icon is associated to the element according to the status.

There are some issues that are very important to be aware of when using attributes. First it is non possible to have multiple values in one attribute. Second HTML cannot be added to attributes because it will create an invalid structure. The only way to add HTML to an XML structure is within an element. In order to add HTML to elements so that it is readable by programming language that is parsing it and does not break the validation of the XML, we need to add CDATA tags to the element tags.

The HTML can be used to display formatted data into a DOM element in our AJAX application front end.
Now we consider the following example:

```
<item> <b> Project Financial Plan </b> </item>
```

In that manner the nesting HTML tags don't work, because the parser will see these elements as nested or child element of the parent rather than HTML tags. While the following structure will be considered in the right way.

```
<item> <![CDATA[<b>Project Financial Plan</b>]] > </item>
```

The text value Project Financial Plan will display as bold text to the user on the document, by simply targeting an HTML tag using DOM and appending the value with JavaScript's intrinsic innerHTML property or using document.write().

Parsing XML

In the body section of the document we can set:

```
<body>
<a href=”javascript: makeRequest('data.xml',
onXMLresponse);”>xml</a>
&nbsp; &nbsp;
<a href=”javascript: makeRequest('data.js',
onJSONresponse);”>json</a>
<br/>
<hr noshade=”noshade”>
<div id=”loading”></div>
<div id=”header_section”></div>
<div id=”body_section”></div>
</body>
```

so we can parse the response as we would like on the specific request being made. The Response Method is:

```
function onXMLResponse (){
      if ( XHR.readyState == 4 )
       {
         var response=XHR.responseXML.documentElement;
         //Parse here
       }
}
```

We will start by parsing the category values from the XML file and adding them to the body div via the innerHTML property.
In the parsing we will use the Javascript's intrinsic *getElementByTagName* method. Using this method will return an array of all elements by the name that you specify without looking at the depth in which they reside.

```
// Categories
document.getElementbyId(“header_section”).innerHTML
=“<b>Agenda</b><br/>”;

var categories = response.getElementByTagName('category');
for ( var i=0; i<categories.length; i++)
{
  window.document.getElementById(“body_section”).innerHTML+=

response.getElementByTagName('category')[i].firstChild.data+”</b
r>”;
}

// Items
var row=response.getElementByTagName('row');
for(var i=0; i<row.length; i++)
{
  var
action=response.getElementByTagName('items')[i].getAttribute('ac
tion');
```

```
  var icon
=response.getElementByTagName('items')[i].getAttribute('icon');

  window.document.getElementById("body_section").innerHTML+=
  action+"<br/>"+icon+"<br/>";

  var items
=response.getElementByTagName('items')[i].childNodes;
  for(var j=0; j<items.length, j++)
  {
    for(k=0; k<items[j].childNodes.length; k++)
    {
     window.document.getElementById("body_section").innerHTML+=
     items[j].childNodes[k].nodeValue+"<br/>";
    }
  }
}
```

Parsing JSON

JSON or *Javascript Object Notation* is a data-interchange format, even if it is not a standard, it is becoming widely accepted. It is essentially an associative array or hash table. JSON parsing is natively with JavaScript's *eval*[23] method, which makes it extremely simple to parse when using it in your AJAX application. The downfall is that the parsing can be quite slow and insecure due to the use of the eval method. Rogue sites can engage in

[23] The eval() function evaluates or executes an argument. If the argument is an expression, eval() evaluates the expression. If the argument is one or more JavaScript statements, eval() executes the statements. For example the following script

```
<script type="text/javascript">
     eval("x=5;y=25;document.write(x*y)");
</script>
```

produces as output 25. Using eval() it is a very simple way to parse JSON text, here is an example

```
<script type="text/javascript">
     var jsontext = "{a1: 'value 1', a2: 'value2'}";
     var object = eval("(" + jsontext + ")");
</script>
```

JavaScript hijacking by sending responses that contains malicious executable code in place of (or hidden inside) JSON Data.

The structure of a JSON file is representative of a JavaScript object in the way that one file can consist of multiple objects, arrays, strings, numbers, and Booleans.

Here is an example of a complete JSON file:

```
{
   "data":
   "categories":
    {
   "category": ["Priority", "Object", "When", "Expiry Time", "Where"]
    },
    "row":
     {
        "items":
         [
              { "action": "alert('Meeting');"
              "icon" : "img/warn.gif"
              "item" : ["<u>Hight</u>",
                       "<b>Project Financial Plan </b>",
                       "2009-09-06 15:30:00",
                       "3",
                       "Purple Room "]
              },

             { "action": "alert('Meeting');"
              "icon" : "img/warn.gif"
              "item" : ["<i>Normal</i>",
                       "<b> Project Management </b>",
                       "2009-10-12 10:30:00",
                       "2",
                       "White Room "]
              },

         ]
     }
}
```

As you can see, it is much slimmer than the XML version of the lack of redundancy in tag names.

In order to parse the data, we will begin by creating the callback method, checking the ready state of the request, and evaluating the `responseText`.

```
Function onJSONResponse()
{
  if ( checkReadyState(XHR,'complete') == true )
  {
    eval(“var response = ( “+XHR.responseText+”)”);
  }
}
```

Let's start by targeting from the data and appending them to the body div.

```
// Categories
   for (var i in response.data.categories.category)
   {

   document.getElementById(“body”).innerHTML+=
   response.data.categories.category[i]+”<br/>”;
  }
```

As you can see, it is very easy to target the data it is parsed into JavaScript object. Property values are accessible by simply using dot syntax to target them by the proper path. The we can simply do for in loop to target all the property values within a specific object.

```
  for (var i in response.data.row.items)
   {
      for (var j in response.data.row.items[i])
      {
        document.getElementById(“body”).innerHTML+=
        response.data.row.items[i][j]+”<br/>”;
      }
    }
```

4.5 EASY AJAX INTERACTIVITY WITH jQuery

Web application interactivity is enhanced by using the jQuery library which allows to write code in a more readable way and enormously simplifies the life to the web developer.

The jQuery function for sending AJAX request is $.ajax(). It is called without a selector because AJAX actions are global functions and are executed independently of the DOM.

The $.ajax() method accepts as an argument only an object containing settings for the AJAX call. It this function is called without any settings the method will load the current page and will do nothing with the result.

Considering the main and more used settings, the object passed as argument to $.ajax() has the following structure:

```
Var AJAXSettings = {};
 1)     AJAXSettings.data =
 2)     AJAXSettings.dataFilter =
 3)     AJAXSettings.dataType =
 4)     AJAXSettings.error =
 5)     AJAXSettings.success =
 6)     AJAXSettings.type =
 7)     AJAXSettings.url =
$.ajax(AJAXSettings);
```

1) the data property describes any data to be sent to the remote script either as a query string `“var1=val1&var2=val2& …”` or as JSON format (`{ “var1” : “val1, “var2”: “val2”, …}`).
2) `dataFilter(data, type)` is a callback function that allows to filter the data coming from the remote script. The function takes two arguments: the raw data returned from the server, and the dataType parameter.
3) `dataType:` this described the type of data expected from the request. If this property is not specified, jQuery will try to get the

result type using the MIME type of the response. The available types are: “xml”, “html”, “script”, “json”, “jsonp”,, and “text”.

4) `error(XMLHttpRequest, textStatus, errorThrown)` is a callback function which is execute in case of request error. The second parameter of the function is a string describing the type of error that occurred. The possible values for the second argument are null, “timeout”, “error”, “notmodified” and “parsererror”.
5) `success(data, textStatus, XMLHttpRequest)` is a callback function that is executed if the request completes successfully. The parameters of this function are: the data returned from the server, formatted according to the 'dataType' parameter; a string describing the status; and the XMLHttpRequest object.
6) type is a string which is the type of request to send. The possible values area GET (the default value), POST, PUT and DELETE.
7) url is the URL to which the request is to be sent.

Let us give an example to show how is easy to use AJAX with this method.

```
Var AJAXSettings = {};
AJAXSettings.type     = “POST”
AJAXSettings.url      = “GetData.php”;
AJAXSettings.data     = “Set=Yes&Yellow=Yes&Red=No”;
AJAXSettings.success  = function (data){
                         $(“#ResultPanel”)
                           .css(“background”,”yellow”)
                           .html(data);
                        };
$.ajax(AJAXSettings);
```

4.6 Modern Web Application: REST

The advent and the diffusion of the AJAX technology have brought to the development of a new approach to the web application design. This new model is called REST. It stands for *REpresentational State Transfer* and it comes from Roy Fielding’s PhD dissertation published in 2000.

Fielding analyzed all networking resources and technologies available for creating distributed applications and arrived to define the following constraints that identify a RESTful system:

- it must be a client-server system;
- it has to be stateless: each request should be independent of others;
- the network infrastructure should support cache at different levels;
- each resource must have a unique address and a valid point of access;
- it must support scalability.

These constraints don't impose what kind of technology to use and, what is more important, we can use existing networking infrastructures such as the Web to create RESTful architectures.

Fielding defines REST in [4.5] as "*a coordinated set of architectural constraints that attempts to minimize latency and network communication while at the same time maximizing the independence and scalability of component implementations. REST enables the caching and reuse of interactions, dynamic substitutability of components, and processing of actions by intermediaries, thereby meeting the needs of an Internet-scale distributed hypermedia system.*"

4.7 REST: ARCHITECTURAL ELEMENTS

REST considers three classes of architectural elements:

- data elements;
- connecting elements (connectors);
- processing elements (components).

Data elements

- *Resource*: it is the key abstraction of information. Any information that can be accessed and transferred between clients and servers is a "resource". A resource can change overtime while its semantic is static. In this manner we refer to a concept instead of a single representation, as a resource may have multiple representations. For example, a resource that represents a circle may accept and return a representation that specifies a centre point and radius, formatted in SVG (Scalable Vector Graphics), but may also accept and return a representation that specifies any three distinct points along the curve as a comma-separated list [4.08].

- *Resource identifiers*: they are used to distinguish between resources. They are the only means for clients and servers to exchange representations. In the web environment the identifier would be an uniform Resource Identifier (URI) as defined in the Internet RFC 2396 [4.09].

- *Representation*: it is what is transferred between the components. A representation is a temporal state of the actual resource located in some storage device at the time of the request. A representation consist of:
    - the content: a sequence of bytes;
    - describing content: representation metadata;
    - metadata describing metadata.

Connectors

REST uses various connector types to encapsulate the activities of accessing resources and transferring resource representations. These connectors could be:

- *Client*: sending requests and receiving responses;
- *Server*: listening for requests and sending responses;
- *Cache*: can be located at the client or server connector to save cacheable responses, can also be shared between several clients;
- *Resolver*: transforms resource identifiers into network address;
- *Tunnel*: relays requests, any component can switch from active behaviour to tunnel behaviour.

The connectors present an abstract interface for component communication, enhancing simplicity and hiding the underlying implementation of resources and communication mechanisms [4.05].

All rest interactions are stateless; as a consequence each request contains all of the information necessary for a connector to understand the request, independent of any requests that might have preceded it [4.05].

Components

REST components are identified by their role within an application.

- *User agents*: uses a client connector (for example a Web Browser) to initiate a request and becomes the ultimate recipient of the responses.
- *Origin server*: uses a server connector to receive the request. It is the definite source for representations of its resources and must be the ultimate recipient of any request that intends to modify the value of resource. Each origin server provides an interface to its services and hides the resource implementation behind this interface.
- *Intermediary components*: they act as both a client and a server in order to forward with possible translation, requests and responses.

Examples of this type of components are proxy and gateway (aka reverse proxy).

4.8 HTTP AND REST

HTTP has a special role in the Web Architecture as both the primary application-level protocol for communication between web components and the only protocol designed for the transfer of resource representation [6.05]. Before describing the architectural component of REST applied to HTTP, we start with a simple application of REST taken from [4.06]. It is a small web service which will provide the following functionalities:

- the user can upload a picture;
- metadata can be attached to pictures;
- pictures and attached metadata can be deleted;
- a list of pictures can be retrieved;
- picture and metadata of a picture can be retrieved.

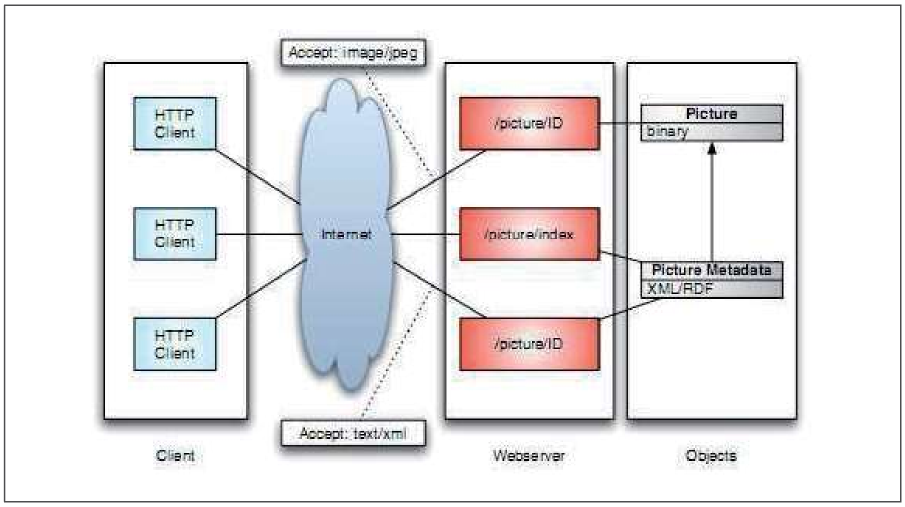

Figure 4.3 – Overview of web service application.

Resources

The resources within the application are:

- Picture;
- Picture-Collection.

Representations

Each resource has associated representation:

- Picture: binary and XML;
- Picture-Collection: XML.

Addressing

The resources are addressable via URI. Only resources can be addressed, not the representations. In fact the client use content negotiation to determine which representation should be returned for example *text/xml* or *image/jpeg*.

Methods

We use the following methods of HTTP:

- PUT is used to upload a new picture to the server;
- POST: is used to append more metadata to the addressed resource;
- DELETE: can be use to delete a resource;
- GET: is used to retrieve a representation of a specified resource.

In general when we applied REST to HTTP we must speak of the following concepts: nouns, verbs, adjectives, meta-data and contents.

- *Nouns*: In HTTP a noun is a URI. It will remain the same and be valid for as long as the web service is on line or the context of a resource is not changed. We use URIs to connect clients and servers to exchange resources in the form of representation. One practise with naming URIs is to remove any non-essential information. We consider for example: http://www.AExampleOfLink.com/login.aspx, the wrong element is the aspx extension. If the web application switches to another system for example PHP, the URI will have to be changed as well.

- *Verbs*: The verb in HTTP is called method. A full list of methods is available in section 9 of RFC 1616 [6.10]. In REST we have constraints on how to manipulate resources. In fact we have four specific actions that we can take upon the resources: Create, Retrieve, Update and Delete (CRUD). A mapping of CRUD actions to the HTTP will be:

Data action	**HTTP protocol equivalent**
CREATE	POST
RETRIEVE	GET
UPDATE	PUT
DELETE	DELETE

- *Meta-data*: In HTTP there are many kinds of meta-data contained in the request and response which could be for example the MIME-types, what program is making the request, what program is running on the server, if the response can be compressed with g-zip etc…
- *Content*: using HTTP to communicate, we can transfer any kind of information that can be passed between clients and servers. For example if we request a Flash movie from YouTube, your browser receive a Flash movie. The data is streamed over TCP/IP and the browser knows how to interpret the binary stream because of the HTTP protocol response header Content Type. Therefore on a

RESTful system the representation of a resource depends on the caller's desired type (MIME type).

4.9 AJAX AND REST

Using AJAX technology and REST together we can design a new framework in which we can take the benefits of both dynamic interactivity provided by AJAX and modern architecture style of REST.

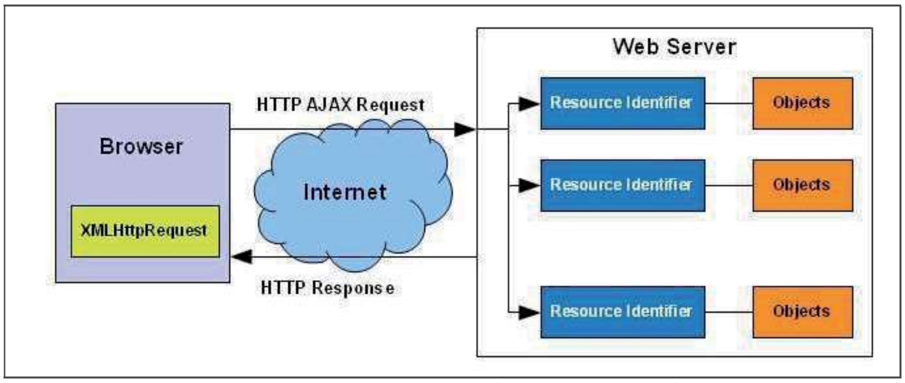

Figure 4.4 – AJAX and REST Framework

The figure 4.4 shows a representation of the aforementioned framework. The main characteristic is that the client end the server are uncoupled. You can create the content on either side indipendently.

The client-side code can provide an infrastructure where the content generated by the resources can be injected into the web page on the browser. Moreover on the client-side you could make use of graphics and innovative representations of the data generated by the resources. In this manner we may simply implement the so-called *Rich Internet Applications*.

On the server-side the objects could consist of flat file as well as a database. The complexity of the object is hidden by its interface. Focusing on the sigle object/resource for example a database, it is easier to optimize it and to increase its access speed.

This framework has got a more important positive aspect. You can use AJAX and REST today, that are exsisting technologies, without throwing out old technologies and replacing them with new ones.

Bibliography

[4.01] **Kris Hadlock**, *Ajax for Web Application Developers*, Sams Publishing 2007;

[4.02] http://www.w3.org/TR/XMLHttpRequest/: the XMLHttpRequest specification defines an API that provides scripted client functionality for transferring data between a client and a server;

[4.03] http://www.json.org: **JSON** (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate;

[4.04] **Luciano Noel Castro**, *Web 2.0, creare siti di nuova generazione*, Sprea Editori S.p.A. 2008;

[4.05] **Roy T. Fielding, Richard N. Taylor**, *Principled Design of the Modern Web Architecture*, ACM Transactions on Internet Technology 2002;

[4.06] **Michael Jakl**, *REST REpresentational State Transfer*, University of Technology Vienna;

[4.07] **Alan Trick**, *An overview of the REST Architecture*, Advanced Web Programming, 2007;

[4.08] http://en.wikipedia.org/wiki/Representational_State_Transfer, from Wikipedia;

[4.09] T. Berners-Lee, R. Fielding, and L. Masinter, *Uniform resource identifiers (URI): generic syntax,* Technical Report Internet RFC 2396, IETF, 1998;

[4.10] "*Hypertext transfer protocol – http/1.1",* RFC 2616, IETF, 1999;

[4.11]	http://jquery.com/, jQuery JavaScript library;