

RWF

THE RICH WEB-CLIENT FRAMEWORK

White Paper

Accompanying Release

Beta 5.0

Martin Hunter

August 2006

BACKGROUND

In the spring of 2001, my work partners and I started using the Microsoft XMLHTTP and XMLDOM ActiveX Objects, XSLT and DHTML to produce feature-rich Web application user interfaces (UIs) for use with Internet Explorer (IE) version 5.5. At that time, IE5.5 was one of the only browsers on the market to embed all of these capabilities, and to also be in common use on the Internet¹.

The core principles of the framework were simple yet effective:

- Use an XMLHTTP instance to perform HTTP post and get transactions with an ASP (classic) page to pass XML documents between the client browser and the web server.
- Transform the XML to HTML using XSL and client-side JavaScript.
- Dynamically update fragments of a pre-loaded *container* HTML page with the transformed HTML (a process referred to as *rendering*).
- Provide a scripted framework for data binding, event handling, data validation and page history.

The idea was to circumvent the excessive (and unnecessary) quantity of HTTP traffic generated by alternative contemporary web frameworks, and to provide a smooth, rich and continuous user experience. This was achieved by transacting only the minimum data required between the client browser and web server, and by dynamically updating HTML on-screen without page refreshes.

Since its inception, the framework architecture and libraries have been iterated, standardised and rationalised. The most recent research version of the framework is the Rich Web-Client Framework (RWF), beta 5.0. It incorporates functionality to display pop-ups, utilise SOAP-based XML Web Services to exchange data with application servers and, importantly, to separate presentation logic concerns using a variation of the MVC design pattern.

Today, some of the techniques used in the RWF have been paralleled and popularised by the AJAX (*Asynchronous JavaScript and XML*) technologies. AJAX is a term that was first coined by developers using JavaScript and XMLHTTP objects to perform asynchronous partial post-backs; a process of exchanging XML-based data between client-side script and XML Web Services without asking the browser to refresh the page in full. The RWF differs in that it utilises *synchronous* transactions with Web Services, though the model can easily be extended to include asynchronous transactions.

Recently, Microsoft have jumped on the AJAX band-wagon with the release of ASP.NET codename "Atlas". Atlas includes extensive client-side script libraries that not only provide core AJAX capabilities but also, through JavaScript object prototyping, extend the ECMAScript (JavaScript) object model to provide pseudo class-oriented programming features such as interfaces and inheritance. The result is a powerful framework for the development of feature-rich user interfaces that deliver a smooth user experience via a variety of common browsers.

Internet Explorer 7 now provides native JavaScript objects for handling HTTP gets and posts (*XmlHttpRequest*), as do other common browsers such as Mozilla FireFox. This extends the reach of AJAX methods on the Internet, and vastly improves the ease by which AJAX based web presentations can be developed and deployed. However, the RWF makes use of Active-X objects other than XMLHTTP. RWF is therefore currently still targeted solely at the Internet Explorer browser audience.

¹ IE 6.0 beta 1 was released in March 2001, though was not commercially viable until August 2001.

INTRODUCTION

The RWF architecture uses AJAX-style techniques for transacting XML-based data between the client browser and web server together with an implementation of the Model / View / Controller (MVC) pattern for managing presentations and reacting to user input. The RWF uses JavaScript classes, CSS behaviours and XSLT to create and dynamically update the HTML displayed by the browser.

The most recent version of the RWF is available for download from:

<http://www.martinhunter.co.nz/research.html>

STARTER KIT

WHAT IS THE RWF?

The RWF is a suite of components on which presentation tiers of web applications using Microsoft IE 5.5 or above can be developed and deployed. The RWF consists of components which offer a variety of generic functionality, including:

- Management of the presentation and display of forms (or "views").
- Validation and verification of form data.
- Persistence of form data.
- Management of application workflow.
- Centralised mechanism for managing communication with the server side application business tiers.

The RWF manages application complexity through:

- The provision of a set of generic components which can be configured for specific application requirements via XML based configuration files and JavaScript constants.
- The provision of interfaces through which bespoke components can be bolted-on and utilised in a consistent and generic fashion.

WHAT COMPONENTS MAKE UP THE RWF?

RWF components are categorised as follows:

- **Presentation (views)**

Presentation components involve the preparation and display of application forms, or 'Views'. These include the StandardView JavaScript class, any *bespoke View* JavaScript classes programmed specifically for application using the IView interface and the StandardPopup JavaScript class and HTML document.

- **Connectivity**

Connectivity components manage communication between the *presentation* and *business* tiers of the application. These include the RWF's `System.ConnMgr`, `SoapRequest` and `SoapResponse` JavaScript classes together with bespoke SOAP XML Web Services.

- **Data**

Data components manage data within the presentation tier. These include the `Xml.XmlManager`, `Xml.XmlDocument`, `Collection`, `System.SessionCache` and `System.AppConfig` JavaScript classes.

- **Validation**

Validation components provide mechanisms for data validation and binding. The `System.Validation` JavaScript class performs data validation, and can be utilised via the `Validation.htc` component (which provides a generic mechanism for validating form fields) and directly in your JavaScript code.

- **Workflow**

Workflow components manage the flow of user activity through the application. Currently, the RWF uses two functional workflow components: The `Page.MainController` and `Page.History` JavaScript classes.

WHAT TECHNOLOGIES ARE EMPLOYED IN THE RWF?

The RWF is based on a number of web oriented technologies on both the server and client sides:

- **HTML** is used by the RWF to present forms. HTML is both static and generated dynamically by *presenter* classes.
- **Cascading Style Sheets (CSS)** is used to decouple the display of HTML components from the structure of an HTML form.
- **JavaScript** is used to program complex logic. JavaScript classes are used extensively and form the basis of most of the RWF components.
- **XML** is used for both the transmission of data between the presentation and business tiers, for storage of data on the client side and for the storage of configuration data.
- **XSLT** is used for the transformation of XML into XHTML for use in the presentation of a form.
- **HTC** is used to manage the behaviour of HTML controls on a form, and to facilitate generic mechanisms such as navigation and field validation.
- **SOAP XML Web Services** are used to exchange XML based data between the RWF presentation components and the business tiers of an application.

BASIC PRINCIPLES IN THE RICH WEB-CLIENT FRAMEWORK

JavaScript CLASSES

To improve the separation of concerns in client-side JavaScript script, the RWF puts JavaScript functions to work as pseudo classes, or *JavaScript classes*. JavaScript classes are JavaScript object “templates” from which new objects can be dynamically created in the scripting engine’s memory at runtime. A function in JavaScript can be treated in a similar way to a class in Java or C#: It can embed variables (data) and other functions (behaviour) in the manner of a class, such that we can *define* an object *type*.

To get a better idea of how a JavaScript class is programmed, let’s look at an example. The following JavaScript class represents a Book. The class consists of a single top-level function, the *class function*, called `Book`. Inside the class function we declare member variables (`var` keywords) and operations (`function` keywords) in a manner similar to the declaration of class members in common OOP languages.

```
Book = function(aName)
{
    // Public operations
    this.Open = Open;

    // Private data members
    var theName = null;

    // <<constructor>>
    Construct(aName);
    function Construct(name)
    {
        this.theName = name;
    }

    // Behaviour
    function Open()
    {
        // Book opening logic
    }
}
```

The `Book` class takes a single variable `aName`, which needs to be passed to it on construction. Though it is not strictly required, we have also included a `Construct()` operation which encapsulates the behaviour to be executed when a new copy of the `Book` is constructed. The code declaring the variable `theName` and calling `Construct()` is executed when the function class is copied. Notice that neither the variable `theName` or the `Construct()` operation are declared in the “public operations” section of the `Book` type. This is because we want these to be scoped as *private* within the `Book` class. In fact, all variables and functions declared within a JavaScript class default to *private* scoping unless the explicitly declared as *public* through use of the `this` keyword.

Client code that uses the `Book` class to create a new `Book` instance will look very familiar to programmers used to C-based OOP languages:

```
var myBook = new Book("Ajax");
```

This line will return a new *copy* of the `Book` object in memory and assign it to the variable `myBook`.

Classes in JavaScript and Java compared

The difference between a JavaScript class and a class in Java is that a JavaScript class is actually a real *object* in memory; it is a *function*. A Java class is *template* from which a real object can be *instantiated* in memory.

In JavaScript, all things, including functions, are arrays. When we use the `new` keyword in JavaScript, we are instructing the scripting engine to take a *copy* of the array (the object) in memory and return a reference to that copy. In Java, the `new` keyword instructs the runtime to *create* a new *instance* of the class (an object), and return a reference to it.

In this paper, when we use the terms *create*, *construct* or *instantiate* with respect to a *JavaScript class*, we are referring to the use of the `new` keyword to create a copy of the JavaScript class object in memory. Though the mechanisms are very different, for simplicity you can think of constructing a JavaScript class as meaning the same thing as constructing a class in Java.

However: A word of warning. Because we *copy* an existing object in JavaScript, rather than *instantiating* a new one from a class, the object's current *state* is also copied. This means that the state of the newly copied object may not be as you might expect. You should ensure that the construction code (the local `var` declarations and `Construct()` function) of your JavaScript class sets *all* local member variables with appropriate values (in the case of `Book`, `theName` is initially declared as `null` and then set to the value of the `aName` parameter in the `Construct()` operation). Don't rely on values that may have changed during the course of the source JavaScript class's lifetime.

JavaScript NAMESPACES

Namespaces can be used in JavaScript in a similar way to C++ or C#. The difference is that you need to declare a namespace as a *function*, which you then declare classes (more *functions*) as being part of.

```
// Namespace "Library"
Library = function() {}

// Class "Library.Book"
Library.Book = function(aName)
{
    // Implementation here!
}
```

The Microsoft ASP.NET Atlas libraries allow you to declare a namespace in a slightly different way, using the `Type.registerNamespace()` operation from the Atlas script libraries.

JavaScript INTERFACES

As we have seen, JavaScript provides extensive support for using functions as templates for objects in a similar manner to *classes*. But, what support is there for class and interface inheritance?

There are a number of mechanisms available to facilitate inheritance between *objects* in JavaScript. The Microsoft ASP.NET codename "Atlas" libraries employ a clever mechanism that utilises prototyping of the base `Function` object in JavaScript to provide a programming model that is very close to true OOP languages².

The model employed in the RWF is somewhat simpler in principle and makes use of some ideas from the *decorator*³ design pattern. In RWF, an *interface* is a JavaScript class that defines a set of public operations. Classes that *implement* the interface declare these same operations as public, together with any others that they need to provide. A naming convention is employed for the implementing class's interface operations such that the interface name is prefixed to the operation name. For example, an operation declared by an *interface* as `MyOperation()` is declared by *classes* that implement that interface as `IMyInterface_MyOperation()`.

To communicate with an implementing object via the interface, client code creates a new *instance* of the interface class and passes it a reference to the implementing object. When an interface operation is called by client code, the interface object simply forwards the call to the relevant operation on the implementing object. This method of delegation from interface to class means that relationships are structured in accordance with the decorator pattern. Logically, however, the interface class does not perform the same role as a decorator since it does not add any logic of its own.

Let's take a look at an example using the `Book` class from earlier and a new interface, `IReadable`:

The interface

```
IReadable = function(readable)
{
    // Public operations
    this.Read = Read;

    // Reference to the implementing object
    var theReadable = readable;

    // Operation definition for Read
    function Read()
    {
        // Read the readable object!
        return theReadable.IReadable_Read();
    }
}
```

Of course, we'll need to extend our `Book` class to incorporate the `Read()` operation defined in the `IReadable` interface:

² See **Laurence Moroney**, "Foundations of Atlas: Rapid AJAX Development with ASP.NET 2.0", Apress (2006) for further information.

³ Decorator is a design pattern from the Gang-of-Four (Gamma, Helm Johnson & Vlissides). See "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley (1995) for further information.

The implementing class

```
Library.Book = function(aName)
{
    // Public interface operations
    // IReadable implementation
    this.IReadable_Read = IReadable_Read;

    // Public class operations
    this.Open = Open;

    // Private data members
    var theName = null;

    // <<constructor>>
    Construct(aName)
    function Construct(name)
    {
        this.theName = name;
    }

    // Behaviour
    function Open()
    {
        // Book opening logic
    }

    // IReadable implementation
    // -> Read operation
    function IReadable_Read()
    {
        return "Once upon a time...";
    }
}
```

To use a Book object via the IReadable interface, client code might look like this:

```
var readable = new IReadable(new Library.Book("Ajax"));
var text = readable.Read();
```

If we wanted to access other operations of the Book object, we would need to create the Book object first, assign it to a variable and then pass it to the new interface instance:

```
var book = new Book("Ajax");
book.Open();
var readable = new IReadable(book);
var text = readable.Read();
```

CLIENT-SIDE XML LIBRARIES

To facilitate the exchange of data between client-side script running in the browser and server-side code, the RWF makes use of the Microsoft XML Active-X (COM) libraries, under the COM library name “*MSXML*”.

Prior to the release of Internet Explorer version 6.0 in August 2001, the MSXML COM library needed to be installed as an Active-X plug-in. This limited the usefulness of the RWF since it required the user to independently install MSXML on the client machine and configure the browser to allow access to it. Whilst this was fine for a controlled corporate environment, it limited the audience that RWF could reach using default settings. Since version 6.0, MSXML has come bundled with the IE browser as standard. This has facilitated the use of XML Active-X objects, and thus the RWF, by a much wider audience.

In Internet Explorer version 7.0 a specific component of the MSXML library used by most AJAX libraries, *XMLHTTP*, has been rolled out as a native JavaScript object called *XmlHttpRequest*. *XmlHttpRequest* is now a W3C standard and is available in other common browsers such as Mozilla FireFox. The use of *XmlHttpRequest*, rather than the *XMLHTTP* Active-X object, vastly enlarges the browser audience that can be reached by the RWF and AJAX. It also eases development, since we no longer need to develop code targeting specific browsers.

Implementing XML in the RWF

In general, the RWF uses two key types from the Microsoft XML libraries:

1. *XMLHTTP*, which is used to facilitate the exchange of XML between client-side script running on the browser and a web server.
2. *DOMDocument*, which is used to store, manipulate and transform XML on behalf of client-side script.

The RWF also makes occasional use of the XML node (element and attribute) types from the Microsoft XML libraries.

All of the MSXML library Active-X objects are wrapped using XML handling JavaScript classes in the RWF. These classes reside in the *xml.js* script file and are considered in depth later in this paper.

MVC IMPLEMENTATION

Overview of the MVC design pattern

Model / View / Controller (MVC) is a software design pattern that divides the application data, screen presentation and user interaction control logic of a system into three separate components.

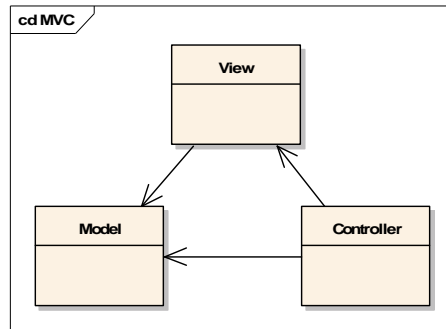


Figure 1: The MVC pattern

The **model** component represents the object, or set of objects, in the application domain. Commonly, we may think of these as the program objects that we create in our software systems to represent logical entities in the business domain. More broadly, the model constitutes objects that embody the purpose of the system in terms of data and behaviour, as opposed to objects that exist purely as framework components that facilitate the operation of the system.

The **view** component presents data from the model on-screen for the user to consume and interact with. Examples of views might be rich Graphical User Interface (GUI) forms or web pages which present data for users to see and manipulate.

In MVC, the view is responsible for ensuring that its presentation reflects the state of the model at all times; when data in the model changes, the view must adjust its presentation to suit. This is often realised by an implementation of the *observer* design pattern, in which each view registers an interest in (or *subscribes* to) changes to the model. When the model is altered, it informs the views registered with it that this has happened, which in turn allows each view to update itself appropriately.

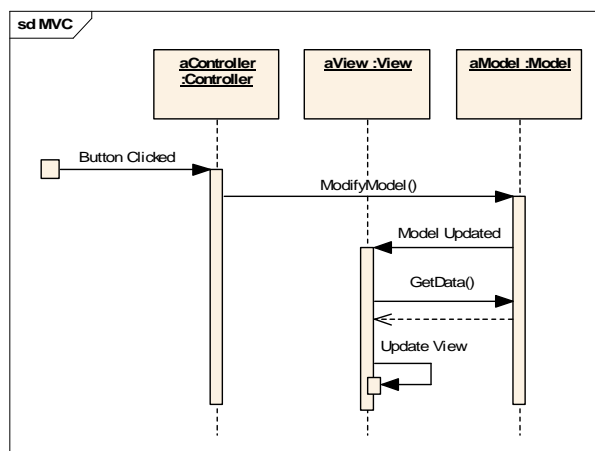


Figure 2: MVC interactions

The **controller** component defines and manages the way the view reacts to input from the user. When a given event “happens” to a view, it is the controller that catches the event and decides what happens next.

Implementation of the MVC pattern in the RWF

Due to the restrictions of the technologies used by the RWF, the framework implements a variation of the MVC pattern. *Controllers* are implemented using JavaScript classes and XSLT, *models* are XML DOM Documents and *views* are fragments of the HTML presentation running in the browser.

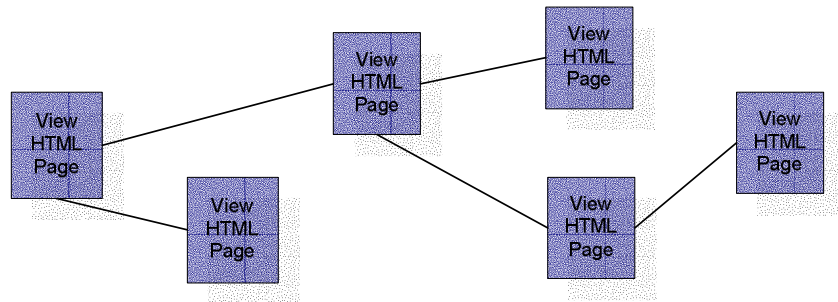
In the RWF, controllers are responsible for assembling presentations, reacting to user input, performing updates on the model and managing communications with the web server. Views are responsible for displaying data, forwarding user events to controllers, and performing updates on the model. The model is responsible only for storing data in a structured way and accepting modifications to data from views and controllers.

In the sections that follow, we’ll be taking an in-depth look at the mechanisms that the RWF uses to facilitate the interactions outlined above.

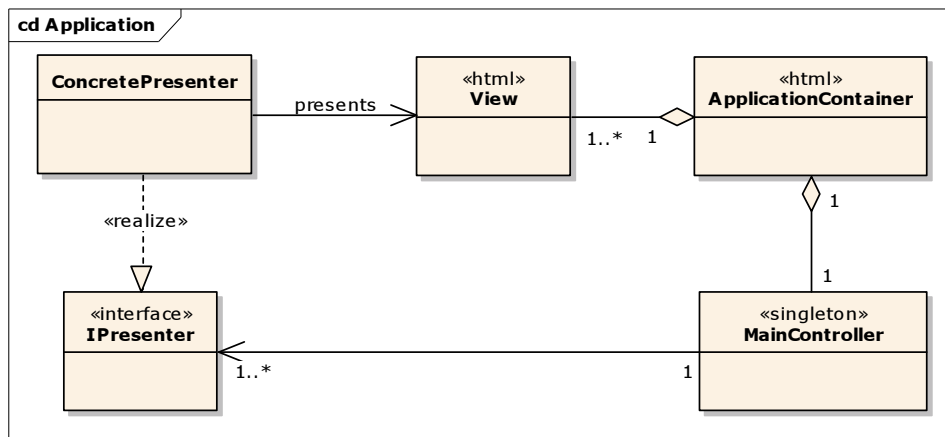
THE RWF FRAMEWORK LIBRARY

THE MAIN PAGE: *Application.htm*

Traditionally, web applications involve the use of a variety of separate HTML documents, which together make up the *views* of the application. The following illustrates a typical application, with a number of HTML pages and their associations:



Applications written using the RWF work differently, in that they consist of *one* generic HTML page (*Application.htm*, referred to within the RWF as the application *main page*) which acts as a *container* for the application views.



Once loaded, the application main page remains for the duration of the application session, and does not get unloaded. The views that make up the application are constructed and displayed dynamically at runtime by code processed within the application main page.

PRESENTER CLASSES AND THE IPresenter INTERFACE

The overall process of instigating dynamic construction and display of application views is managed by the RWF's presentation workflow controller JavaScript class, *Page.MainController*⁴. *Application.htm* hosts a singleton instance of *MainController* which coordinates a set of *presenter* JavaScript classes. Each presenter is responsible for providing an HTML presentation of itself for display on-screen.

A *presenter* is any JavaScript class that implements the *IPresenter* interface. The *IPresenter* interface defines a set of operations that the RWF requires presenter

⁴ The *Page.MainController* class is covered in detail later in this paper.

classes to provide in order to implement presentation rendering, data binding, automatic data persistence and page histories amongst others.

The following JavaScript sample illustrates the `IPresenter` in full. Note that the interface code follows the *decorator* design pattern: `IPresenter` forwards calls made to its operations through to implementing operations on the concrete presenter that is passed to the interface on construction.

```
IPresenter = function(aPresentable)
{
    // Public Interface ops
    this.GetPresentation = GetPresentation;
    this.Persist = Persist;
    this.Finalise = Finalise;
    this.UpdateField = UpdateField;
    this.TypeOf = TypeOf;
    this.HistoryLabel = HistoryLabel;
    this.PresenterRef = PresenterRef;

    // Decorated instance
    var thePresentable = aPresentable;

    // Decorator Ops
    function GetPresentation()
    {
        return thePresentable.IPresenter_GetPresentation();
    }
    function Persist()
    {
        thePresentable.IPresenter_Persist();
    }
    function Finalise()
    {
        thePresentable.IPresenter_Finalise();
    }
    function UpdateField(strXPath, varValue)
    {
        thePresentable.IPresenter_UpdateField(strXPath, varValue);
    }
    function TypeOf(anObjectType)
    {
        return thePresentable.IPresenter_TypeOf(anObjectType);
    }
    function HistoryLabel()
    {
        return thePresentable.IPresenter_HistoryLabel();
    }
    function PresenterRef()
    {
        return thePresentable;
    }
}
```

`IPresenter` resides in the *System.js* framework file. It is not, however, defined as being within the `System` namespace so it should be referenced directly in client code. The following sample illustrates:

```
var myPresenter = new IPresenter(new MyConcretePresenter());
```

The NavigationPresenter Class

The RWF defines just one concrete presenter that is *required*: `NavigationPresenter`. This resides in the `application/presenters` folder, and is responsible for providing an HTML presentation to display in the `application.htm`'s `divNavigationTarget` HTML `<DIV>` tag. The `NavigationPresenter` is automatically loaded by the `MainController` class when the RWF is first started, and its `GetPresentation()` operation is invoked to generate a navigation panel for display.

Bespoke Presenter Classes

Bespoke, user-defined presenters need to be added to the RWF to provide for the display and behaviour of your application's views.

A bespoke presenter works in exactly the same way as the `NavigationPresenter`: It implements the `IPresenter` interface, through which the `MainController` can invoke behaviour desired to get an HTML presentation for display and to react to user input.

To add a new presenter, you must undertake the following:

1. Create a new *presenter* JavaScript class. This must reside in a JavaScript (`.js`) file within the `application/presenters` folder structure. For example, in sample code that accompanies this paper, a number of Book related presenter classes have been added in the `application/presenters/Book` folder.
2. Create any required XSLT for use by the presenter JavaScript class to transform XML data from a Web Service into an XHTML presentation. This will need to be loaded and used explicitly within your presenter class code.
3. Add a user-defined public operation to the `Page.MainController` class to perform appropriate logic to show your new presentation and to hook it into the RWF to allow it to receive and react to user input.

The following code sample illustrates how a presenter might be structured. Note that the class uses a locally declared instance of `Xml.XmlManager` to transform XML data from a Web Service into an XHTML presentation in the `IPresenter_GetPresentation()` operation implementation. Note also that this operation makes use of the `InvokeWebService()` operation on the RWF's `System.ConnMgr` class instance (we'll be covering this class in detail later in this paper).

```
// =====  
// Class MyConcretePresenter  
// Implements: IPresenter  
// =====  
MyConcretePresenter = function()  
{  
    // IPresenter implementation  
    this.IPresenter_GetPresentation = IPresenter_GetPresentation;  
    this.IPresenter_Persist = IPresenter_Persist;  
    this.IPresenter_Finalise = IPresenter_Finalise;  
    this.IPresenter_UpdateField = IPresenter_UpdateField;  
    this.IPresenter_TypeOf = IPresenter_TypeOf;  
    this.IPresenter_HistoryLabel = IPresenter_HistoryLabel;  
  
    // Local data members  
    var theXmlManager = null;  
  
    // <<constructor>>  
}
```

```

Construct();
function Construct()
{
    theXmlManager = new Xml.XmlManager();
}

// IPresenter implementation ops
function IPresenter_GetPresentation()
{
    theXmlManager.Data().LoadXML(
        System.ConnMgr.InvokeWebService(
            "http://www.rwf.org/",
            "GetSomeData",
            null,
            false));
    theXmlManager.UsingStylesheet(
        "application/presenters/SomeTransformCode.xslt");
    return theXmlManager.Transform();
}
function IPresenter_Persist()
{
    // Code to persist data here
}
function IPresenter_Finalise()
{
    // Code to perform any pre-persistence
    // finalisation of state
}
function IPresenter_UpdateField(strXPath, varValue)
{
    // Code to update local XML data with value passed
}
function IPresenter_TypeOf(aType)
{
    return (aType == MyConcretePresenter);
}
function IPresenter_HistoryLabel()
{
    return "MyConcretePresenter";
}
}

```

THE Page NAMESPACE

The `Page` namespace provides access to singleton instances of classes within the RWF that relate to the control of the presentation flow of the application. These include `Page.MainController` and `Page.History`.

Page.MainController

The singleton instance of `MainController` is responsible for controlling the overall flow of presentations within the application. The `MainController` JavaScript class resides in the `workflow.js` file. It contains:

- RWF library code that manages a variety of tasks including displaying Presenters, managing page history and forwarding data to Presenters for updates or persistence.
- User-defined code for displaying ("showing") user-defined Presenters.

If you open and view the `workflow.js` file, you'll find comments indicating that the top-half of the `Page.MainController` class is **not** modifiable. The code in this section is needed for the successful operation of the RWF. In the bottom-half you'll find **TODO:** comments indicating where to put user-defined code for displaying bespoke Presenters and setting up which Presenter is displayed by default when the application first loads.

Adding a new Presenter to display is easy. You will first need to write a function to create a new instance of your bespoke Presenter and ask the `MainController` to display it (via the `IPresenter` interface). Secondly, you'll need to setup this function as public on the `MainController` class.

```
// TODO: Add new Show operations for your Presenters below
// (2) Public operation for BookPresenter
this.ShowBookPresenter = ShowBookPresenter;

// (1) Implementation for ShowBookPresenter
function ShowBookPresenter ()
{
    RenderAsCurrentPresenter(new IPresenter(new BookPresenter()));
}
}
```

Page.History

The singleton instance of `History` is responsible for managing the history of pages visited in the application and providing functionality to allow the user to scroll "back" through pages they have visited. `MainController` incorporates code that uses `History` within the RWF library code section, so it's automatic and there to be used if you need it.

To use page history in your application, you simply need to call the `ShowLastPresenter()` operation on `MainController`. This will present the page that was displayed immediately *prior* to the current one. When you call `ShowLastPresenter()`, `History` manages the page context such that, when you call `ShowLastPresenter()` again, the next page "back" is displayed.

Here's an example:

```
Page.MainController.ShowLastPresenter();
```

THE System NAMESPACE

The `System` namespace provides access to RWF library code and singleton instances of classes that constitute the core of the RWF architecture. In this section we cover a few of the important classes in the `System` namespace.

System.AppConfig

`System.AppConfig` is a singleton instance that facilitates access to application configuration information stored in the `application.xml` configuration file from the `application` folder. From RWF beta 5.0, the application configuration includes the RWF version, application title (displayed at the top of the application page), the default SOAP Web Service URL to be used by the RWF to exchange XML with, and a group of XML nodes and data regarding the security policy of the RWF.

When the application starts, a check is made that the framework version stored in the `application.xml` file matches that expected by `application.htm`. If they differ, the RWF will stop loading the application and warn you that the RWF version is not appropriate.

System.ConnMgr

`System.ConnMgr` is a singleton instance that facilitates the exchange of XML-based data with SOAP Web Services.

By default, `System.ConnMgr` will use the Web Service URL and namespace from the `application.xml` configuration file. However, you can set the URL to be used to something else via the `UseWebService()` operation.

To invoke a Web Service operation, you can use either the:

- `InvokeWebService()` operation, which takes as its parameters an *operation name* and a set of *parameters* (as an instance of a `Collection` class) and returns XML.
- `InvokeVoidWebService()` operation, which takes the same parameters as above but does not return anything.

To see how you might use `System.ConnMgr`, let's look at an example. Consider the following code, which is a concrete Presenter's implementation of the `IPresenter` interface `GetPresentation()` operation from the RWF sample code:

```
function IPresenter_GetPresentation()
{
    theXmlManager.Data().LoadXML(
        System.ConnMgr.InvokeWebService("GetBooks", null, false));
    theXmlManager.UsingStylesheet(
        "application/presenters/Book/BookList.xslt");
    return theXmlManager.Transform();
}
```

In the above sample, the `InvokeWebService()` operation is used to invoke the `GetBooks` operation of a Web Service. No parameters are passed (`null`). The result is loaded into a local `Xml.XmlManager` instance, called `theXmlManager`, which uses an XSLT to transform the XML into an XHTML presentation to return for display by the browser.

Modifying the Default Web Service used by System.ConnMgr

The URL and namespace of the default Web Service used by `System.ConnMgr` are configured in the `application.xml` file in the `application` folder. The file contains an XML element called `<default-web-service>` that is located in the XPath `application-configuration/application/connectivity`. This contains two elements, `<url>` and `<namespace>`, that contain the URL and namespace of the default Web Service respectively.

The following example illustrates the `<default-web-service>` element in the `application.xml` file.

```
...
<default-web-service>
  <url>http://localhost:2785/RWF/Remote.asmx</url>
  <namespace>http://www.rwf.org/</namespace>
</default-web-service>
...
```

System.Encryption

`System.Encryption` is a singleton instance within the RWF that provides functionality to encrypt and decrypt string (text) values, using a variety of encryption algorithms. `System.Encryption` resides in the `system.js` file in the `framework` folder.

`System.Encryption` follows the *strategy* design pattern. Each type of encryption algorithm used by `System.Encryption` is implemented within the RWF as an independent JavaScript class, called an *encryption* class. Each encryption class implements a common interface that defines two public operations: `Encrypt()` and `Decrypt()`. When implemented by an encryption class, these operations encrypt or decrypt a string value using the encryption algorithms coded into the encryption class.

When we want to use a specific type of encryption class, we create an instance of it and pass it to `System.Encryption` via the `SetEncryptionStrategy()` operation. When a call is then made to the `Encrypt()` or `Decrypt()` operations on `System.Encryption`, the call is internally forwarded on to whichever encryption strategy instance we have set.

The following code sample illustrates how we might use `System.Encryption` and the RWF's `MD5Encryption` encryption strategy to perform an encryption of a string value:

```
var unencrypted = "hello world";
System.Encryption.SetEncryptionStrategy(new MD5Encryption());
var encrypted = System.Encryption.Encrypt(unencrypted);
```

Encryption Strategies in the RWF

Two encryption classes are supplied with the as standard. These are:

- The `MD5Encryption`⁵ class, which is a forward-only encryption class that encrypts string values using the MD5 message digest algorithm but *cannot perform decryptions* of these digests.

⁵ The `MD5Encryption` class makes use of a JavaScript implementation of the MD5 algorithm residing in the `md5.js` RWF framework file. The contents of this file is © Paul Johnston 1999 – 2002, and is supplied under a separate BSD licence to the RWF. Please refer to <http://pajhome.org.uk/crypt/md5/> for further information regarding the algorithm and licence.

- The `ClearTextEncryption` class, which performs no encryption.

This basic set of encryption classes can be supplemented by your own, bespoke encryption classes if you want. All you need to do is create a new encryption class that implements the `Encrypt()` and `Decrypt()` operations, and set this as the encryption strategy to be used by `System.Encryption` via the `SetEncryptionStrategy()` operation. If you open the `system.js` file from the RWF *framework* folder, you will find the RWF's `MD5Encryption` and `ClearTextEncryption` JavaScript classes. Should you need, you can refer to these to identify how to structure your bespoke encryption classes.

Use of the System.Encryption within the RWF

The RWF's User authentication and application logon functionality is implemented using `System.Encryption`. The `System.User` class uses `System.Encryption` to *encrypt* a User's entered *password* so that it can be transmitted securely across the Internet to a Web Service that can authenticate the User's username and password details.

By default the MD5 encryption algorithm is used by `System.User` to encrypt the User's entered password. This can be modified by altering the `<default-encryption>` XML element's value in the `application.xml` configuration file in the *application* folder. In Beta 5.0 of the RWF, the value of `<default-encryption>` can be either "MD5" (to use the MD5 encryption algorithm) or "ClearText" (to use **no** encryption).

System.User

`System.User` is a singleton instance in the RWF that is responsible for storing a User's *username* and *password* information and for the authentication of this information using a remote Web Service. `System.User` resides in the `system.js` file in the *framework* folder.

System.User and the LogonPresenter class

`System.User` is invoked automatically by `Page.MainController` when the RWF is started.

When `Page.MainController` is asked to display a presentation, it first queries `System.User`, via the `IsLoggedIn()` operation, to identify whether the User logon information (username and password) has been submitted and authenticated. If not, then `Page.MainController` presents a logon page, via the `LogonPresenter` class, for the User to submit their username and password details. When the User submits these details, `Page.MainController` then calls `System.User.Logon()` to authenticate the username and password information.

The `Logon()` operation on `System.User` uses a Web Service to authenticate username and password information. The Web Service *URL*, *namespace* and *operation* used by `System.User` for this purpose are configured in the `application.xml` configuration file in the *application* folder.

The following example illustrates the section of the `application.xml` configuration file responsible for storing the user authentication (logon) Web Service information.

```
<security user-logon="required">
  <default-encryption>MD5</default-encryption>
  <logon-web-service>
    <url>http://localhost:2785/RWF/Remote.asmx</url>
    <operation>AuthenticateUser</operation>
    <namespace>http://www.rwf.org</namespace>
  </logon-web-service>
</security>
```

If you would like to disable automatic User authentication and logon in the RWF, you will need to set the value of the *user-logon* attribute of `<security>` XML element in the *application.xml* file to "none". To enable authentication, set this value to "required" as illustrated in the example above.

System.SessionCache

The role of System.SessionCache

`System.SessionCache` is a singleton instance that persists name/value paired data in local browser memory. This allows different presenter instances running in the RWF to:

1. Exchange data between themselves.
2. Store a copy of data obtained from a remote source for use later in the user session.

The primary role of `System.SessionCache` is to facilitate the first purpose above. As a side-effect the second is also possible. However, the use of `SessionCache` to store remote data should be considered and constrained. Caching a copy of remote data locally may mean that the copy becomes out of sync with the remote data. As such, you should:

- Use `SessionCache` to store a copy remote data only when strictly required; for example, when you application is running over a very slow communication link with the Web Service from which the data is sourced.
- Apply caution when choosing which data to cache; for example, cache "standing data" which is modified only very infrequently.
- Apply controls in the design and implementation of your code regarding which presenters can use the `SessionCache` to store remote data.

If you are aware of the capabilities and consequences of using `SessionCache` to store a copy of remote data, then it can become a very powerful medium in the improvement of application performance and the smoothness of the user experience.

How to use System.SessionCache

`SessionCache` is a simple class that uses an instance of a `Collection` class to store name/value paired data. `SessionCache` exposes public three operations:

- `Add(object, key)` which adds a name (*key*) and value (*object*) pair into cache.
- `Find(key)` which returns the object associated with the key in the cache. If no object is found, `null` is returned instead.
- `Wipe(key)` which removes object associated with the passed key from the cache.

An example of the use of `SessionCache` can be found in the `BookSearchPresenter` and `BookSearchResultPresenter` JavaScript classes in the sample code that accompanies this paper (in the `application/presenters/Book` folder). `BookSearchPresenter` provides a presentation of a Book search form. When the user enters a search query and clicks the "Go" button, the `BookSearchPresenter` adds the search query into the `SessionCache`, under the key "search-query". It then calls the `MainController` to show the `BookSearchResultPresenter`. When the `GetPresentation()` operation of `BookSearchResultPresenter` is called, the class looks asks `SessionCache` for the object associated with the key "search-query". This is then used in the invocation of a Web Service which performs a database look up using the query sting.

System.Validation

`System.Validation` is a singleton instance that contains functionality to validate data. Public operations exposed by `System.Validation` fit into two broad categories:

- Operations that validate raw data.
- Operations that validate the data contained within the `value` attribute of HTML control elements.

`System.Validation` class resides in the `validation.js` JavaScript file in the *framework* RWF folder. It can be referenced directly by your JavaScript code or automatically through the RWF's built-in validation mechanism. These are covered separately below.

Referencing System.Validation directly

Making use of the `Validation` instance operations is easy. All you need to do is call the appropriate validation operation, passing it a variable the value of which you want to validate. The validation operation will return a `true` or `false` value depending on whether the value passes the validation criteria embedded in the operation (`true`) or not (`false`).

In the following code sample, we use the `Validation` class to validate whether a value passed to it is *numeric* or not. If this code were to be executed, an alert boxes would be shown reading "true" and then "false".

```
alert(System.Validation.isNumeric(1));  
alert(System.Validation.isNumeric("NAN"));
```

Using the automatic validation mechanism in the RWF

The RWF includes functionality for validating HTML control data automatically. This is facilitated using the `validInput` and `validSelect` CSS classes, from the `validation.css` and `validation.htc` framework files, which can be included in the `class` attribute of `<INPUT>` and `<SELECT>` HTML control elements respectively.

When an `<INPUT>` (or `<SELECT>`) HTML control element is given the `validInput` (or `validSelect`) CSS class, behaviour JavaScript code is automatically attached to the element that is invoked when the value of the element changes. This code calls into the RWF validation libraries to validate the control data based on the value of two user-defined HTML attributes on the controls:

- The value of a `VALIDATE` attribute indicates what data type the RWF should be validated as.

- The value of a `MANDATORY` attribute indicates whether the control must contain data for validation to succeed.

The value of the `MANDATORY` attribute can either be "yes" or "no", depending on whether the control data is mandatory ("yes") or not ("no"). The `VALIDATE` attribute can take on a range of values, as summarised in the following table.

VALIDATE Value	Meaning
STRING	Validates the control data as a string of text characters
DATE	Validates the control data as a date
FUTUREDATE	Validates the control data as a date in the future (exclusive of now)
PASTDATE	Validates the control data as a date in the past (exclusive of now)
TIME	Validates the control data as a time
NUMERIC	Validates the control data as a number
CURRENCY	Validates the control data as a currency value
DECIMAL	Validates the control data as a decimal
CUSTOM-PATTERN	Validates the control data as a user-defined pattern
NONE	No validation required

To use automatic validation, you need to program your presenters to incorporate validation in the HTML they generate. The following code sample illustrates how an `<INPUT>` HTML tag needs to be structured to use validation. Note that we have chosen to validate the `INPUT`'s value as a date and that the value must be completed by the user (i.e. it is mandatory):

```
<INPUT TYPE="text" ID="txtTestDate" VALIDATE="DATE" MANDATORY="yes" />
```

User-defined pattern validation

The RWF allows you to define your own regular expressions (patterns) to use in the validation of data. This is done using the `System.Patterns` singleton class instance from the `system.js` JavaScript file in the `framework` folder.

When your application loads for this first time, the `System.Patterns` class must be initialised with any custom data types that you would like to use. This is done in the `Construct()` operation of the `Page.MainController` class, as follows:

```
// Constructor function for MainController class
function Construct()
{
    // Build Custom Patterns
    System.Patterns.AddCustomPattern(/[A-Z]{1}/, "FIXEDSTRING1");
}

```

These patterns can then be referenced in your generated HTML presentation code:

```
<INPUT TYPE="text" VALIDATE="CUSTOM-PATTERN" PATTERN="FIXEDSTRING1" />
```

Data Binding

The RWF incorporates a mechanism for binding data to the value of HTML controls. This is implemented as part of the validation mechanism: Once an HTML control element's data has been validated, the framework validation code calls into the `MainController` to coordinate the update of data using the currently active presenter.

If you open the file `validation.htc` in the `framework` folder and scroll to the bottom of the `validateField()` function, you'll find code that calls the `MainController.UpdateNode()` operation. This operation takes two parameters:

- An *XPath* to the *XML node* (element or attribute) in the current presenter's underlying XML data document that needs to be updated.
- A *value* that the *value of the XML Node* needs to be set to.

The *value* parameter is simply the data contained in the value attribute of the HTML control element being validated. But, where do we obtain the correct *XPath* from?

You will notice from the code in `validation.htc` that we refer to a user-defined attribute, called `XPATH`, which is attached to the HTML control element that is being validated. The `XPATH` attribute is created and attached to the HTML control by the *presenter* when it produces its HTML presentation to be displayed. The value of the `XPATH` attribute can either be hard-wired (if it is known and understood to not vary) or constructed dynamically by the presenter when generates its HTML presentation.

Constructing the XPath dynamically

The RWF includes an XSLT template for constructing values for an `XPATH` attribute which can be included and referenced by the XSLT code that a presenter incorporates to produce HTML presentations. This template is called `buildXPath` and resides in the `xpath.xslt` file in the `framework` folder.

The `buildXPath` template incorporates functionality to build and output a string of text characters that represents an XPath to a *context* XML node which passed to it as the `$oNode` template parameter. XSLT code that calls the `buildXPath` template must pass the context node, `oNode`, and will be returned a string of text characters that represents the *XPath* to that context node in the XML document being processed.

So lets have a look at how the `buildXPath` template can be used. The following code demonstrates how you might write utilise `buildXPath` in your presenter's XSLT templates:

...

```
<xsl:include href="../../../framework/xpath.xslt" />
```

...

```
<INPUT TYPE="text" ID="txtName" NAME="Name" CLASS="validInput textfield
300px" VALIDATE="STRING" MANDATORY="yes" VALUE="{@name}">
  <xsl:attribute name="XPATH"><xsl:call-template
name="buildXPath"><xsl:with-param name="oNode" select="." /></xsl:call-
template>/@name</xsl:attribute>
</INPUT>
```

The XSLT shown above is designed to output an `<INPUT>` HTML control element, with an ID value of `"txtName"` and an attribute called `XPATH`. The value of `XPATH` is set to the result of a call to the `buildXPath` template. The context node passed to `buildXPath`, the parameter `oNode`, is equal to the *current context node* (or `"."` in XSLT short-hand notation). The code shown also appends `"/@name"` to the result of the call to `buildXPath` so that the value of the `XPATH` attribute equals the full *XPath* to the *name attribute* of the current context XML element.

In the above code, you will also notice that:

- The sample incorporates code to include the file `xpath.xslt`.
- The `<INPUT>` element contains the attributes and attribute-values necessary to invoke the RWF's validation framework for this element (i.e. `CLASS` contains the value `"validInput"` and both the `VALIDATE` and `MANDATORY` attributes exist and are given valid values).
- No unnecessary *white-space* (formatting) is included within the `<xsl:attribute>` tags. This is because any white space included in the tags would be outputted as part of the value of the `XPATH` attribute, and this would invalidate the value of `XPATH` as an *XPath*.

THE Xml NAMESPACE

The `Xml` namespace provides access to RWF classes that manage XML documents. In this section we cover a few of the important classes in the `Xml` namespace.

Xml.XmlManager

`Xml.XmlManager` is a general-purpose class for managing data stored in XML documents. Functionality provided includes document loading from local and remote sources, parsing, reading, update and transformation using XSLT. `Xml.XmlManager` is used throughout the RWF framework libraries, and resides in the `xml.js` file in the `framework` folder.

Loading XML using Xml.XmlManager

To load an XML document using `XmlManager`, you simply call one of the *Load** operations on an instance of `XmlManager`. The following code illustrates:

```
// Create a new instance of XmlManager
var xml = new Xml.XmlManager();

// Loads raw XML
xml.Data().LoadXML("<test />");

// Loads XML from a URL
xml.Data().LoadURL("test.xml")
```

Note that the *Load** operations offered by `Xml.XmlManager` automatically detect and report load and parse failures.

Using Xml.XmlManager to transform XML

Once you've loaded an XML document, you can then transform the XML into another XML document by using the XSLT transform operations provided by `XmlManager`.

Following the code sample above, your next steps would be:

```
xml.UsingStylesheet("test.xslt");
var newXml = xml.Transform();
```

These operations would load the remote XSLT template file `test.xslt` and use it to transform the XML document stored by the `XmlManager` instance into another XML document which is returned, as a *string*, into the variable `newXml`.

To get a better idea of how to use the `XmlManager` class to load, transform and update XML data please refer to the `AddBookPresenter.js` file in the `application/presenters/Book` folder of the code sample accompanying this paper.

Xml.XmlDocument

`Xml.XmlDocument` is a general-purpose class for managing a single XML data document. `XmlDocument` is used extensively by the RWF framework libraries and is used internally by `XmlManager` to perform XML document loading and transformation.

Internally, `XmlDocument` uses an instance of the `MSXML.DOMDocument` Active-X object (COM class) to load and manipulate its XML. As such, the `XmlDocument` class

can be seen as a *wrapper* for `MSXML.DOMDocument` that provides a simple and easy-to-use interface for clients within the RWF, but then delegates the actual provision of services to `MSXML.DOMDocument` to perform.

The `MSXML.DOMDocument` instance that `XmlDocument` uses internally can be obtained via the `DOM()` operation of `XmlDocument`. This means that client code can gain access to the underlying `MSXML.DOMDocument` for specific operations if required and is not restricted to the interface offered by `XmlDocument`.

RUNNING THE RWF IN DEBUG MODE: THE `Debug` SINGLETON INSTANCE

The `Debug` singleton instance allows developers using the RWF to output lines to a debug console at runtime, to ease the process of debugging their code. The `Debug` instance can be referenced directly in code, using the `Debug` keyword, and offers two operations `Print()` and `Close()`.

The `Print()` operation allows a developer to output a value to the debug console. The `Close()` operation closes the debug console. The following illustrates:

```
// Prints "my message" to the debug console
Debug.Print("my message");

// Closes the debug console
Debug.Close();
```

To run the RWF in debug mode, simply append the query string "debug" to the URL for *application.htm* when loading your application in a browser. For instance, opening the following URL will launch the RWF in debug mode:

<http://mysite.org/application.htm?debug>