Hyfinity

MVC Application Design Principles

Hyfinity Limited

Innovation Centre

Central Boulevard

Blythe Valley Park

Solihull

West Midlands B90 8AJ

Web: http://www.hyfinity.com

Tel: +44 (0)121 506 9111

Fax: +44 (0)121 506 9112

INTRODUCTION

DOCUMENT ORIENTED PROGRAMMING

XML and Web Services are being utilised in more and more applications, especially within Service-Oriented Architectures (SOA). Processing XML information using traditional programming languages can be cumbersome, time consuming and more difficult to maintain. One of the main reasons for this is the need to 'break-down' XML documents into objects and methods in order to access parts of the documents. For example, if the *customer id* is required and this field resides 'deep' within the XML document then the document has to be parsed and the information retrieved in order to make decisions, based on this information element. Some of the drawbacks of this approach include:

- The document context is lost when the XML information is 'broken down' into objects
- Accessing information using 3GLs is difficult to program and maintain
- Processing XML information in 3GL can remove focus for developers from the application into the more mundane technology aspects that are required to process and orchestrate XML information

XPath was specifically created for efficient access to information within XML documents. Due to the distributed nature of SOA and the large amounts of XML information processing, a different approach is required for building such applications.

HYFINITY'S MORPHYC ARCHITECTURE

Hyfnity's Morphyc Architecture is designed to process XML information, without distorting the document concept. Using its powerful **XPath-based native XML Rules Engine** it is possible to process and orchestrate XML information at source, without the need for complex programming.

MVC OVERVIEW

MVC is a Morphyc application and is based on the classic **Model-View-Controller architecture**. MVC provides RAD capabilities that can reuse XML information assets to rapidly construct distributed applications. MVC uses an assembly approach to reduce the amount of programming required and automates most of the manual tasks associated with SOA applications.

- **Auto generation of web service proxies** from WSDL files. This removes the need to manually create remote web service calls and construct SOAP wrappers.
- **Auto generation of web screens** from WSDL and XSD information, providing accelerated development of web screens.
- **Auto generation of client-side scripts** for data validation.
- **Auto binding of screen information** to server-side XML data structures, removing the need for programming.
- **Auto generation of basic orchestration** logic to enable end-to-end transactions.

- **Powerful, native XML Rules Engine** for processing XML documents, without the need for traditional programming.

PURPOSE

This overview document provides an introduction to distributed applications, residing within SOA environments. It also provides detail on key design components and the considerations for the design of such systems.

The document covers how the XML assets created during the design process can be used by MVC to rapidly construct complete applications.

This is a high-level design document and assumes the reader is knowledgeable in XML and related technologies. Please consult the MVC documentation for more detailed information on MVC application development.

COMPONENTS OF A DISTRIBUTED XML AND WEB SERVICES APPLICATION

APPLICATION DESIGN

A typical distributed application architecture consists of the following elements.

- **Web Services that 'wrap' remote systems**. These systems may be in-house or reside within external organisations. These web services will be described using WSDL files.
- **A middle tier that orchestrates** calls to these remote web services to retrieve information and store updated information. This tier will is also responsible for the aggregation and splitting of information. For example, if a web screen requires information from multiple web services then these calls have to be made to aggregate the information before presenting to the screen. The reverse may be required after information has been captured.
- **A user interface-tier** that handles the creation and display of screen information.

MVC has a range of RAD capabilities that can significantly accelerate the development of such systems, removing much of the complexity and also producing repeatable and templated code for screens, server-side validation and orchestration.

MESSAGE DESIGN (XML INFORMATION MODELLING)

Typically, the following development steps are required for constructing an enterprise application using MVC:

- **Assemble or design the WSDL files** that represent the remote web services that will be called from the screens via the middle-tier orchestration logic. It is very important that these 'contracts' are agreed and version controlled at an early stage to minimise impact on the user interface development.
- **Design the messages** between the middle-tier and the remote web services. Once again, this is very important to ensure minimum impact on the user interface. The message design should consider the following elements:
    - **Wrapper** format for the message. This will typically be SOAP, but there is no restriction within MVC, so long as the message is well-formed XML.
    - **Header information**, such as security credentials or other non-data related elements that are system-level and required by the remote web services.
    - **The main body** (data) part of the message to be used for transporting the application information. Please note that although SOAP wrappers provide a Body tag, it is important to consider a separate tag that is applicable to the application itself, and is well-formed XML. This aids better processing of application information in the Body and avoids situations where information is extracted from the SOAP body, which is not well-formed outside of the SOAP wrapper.

- **Footer information**, which may be contained in the main body. This may contain **error information** that is returned from the remote web services, including the status of the call.

An additional consideration is whether the message formats are similar for the requests and responses to the remote web services, i.e. does the remote web service expect no response section during a request or an empty response during a request. The same consideration applies during a response from the web service.

During message design, update transactions should be paid additional attention. This is because the modified information may not be sent back via the same fields. The changed information may be mapped to different fields when the information is returned to the remote services. This may dictate a different structure during the response within the main body of the data.

Controllers that orchestrate screen information with remote web services will require **additional message design**. This is especially true of controllers that aggregate information from multiple web services calls. The message design will need to show the format of the XML structure within the controllers as more and more information is collated for example.

WEB SERVICE CLONES AND TEST DATA

Once the Message Design in complete, there should be request and response instance messages available for each web service method. This information can be used to setup a 'dummy' remote web service environment that can be very useful for integration testing.

- **Create web service clones(s)** that emulate the remote services. This is especially important if the remote web services are scheduled for parallel build because without the creation of the web service clones it will not be possible to test the application. Provide basic rules for these clones to enable the receipt of web service requests as defined by the WSDLs and the production of responses adhering to the WSDL definitions. This can be achieved by using the instance messages, created during the message design phase.

- **Create basic test data** that can be used by the web service clones. These are essentially request and response messages adhering to the message design and can be generated from the message design schemas.

- **Use a web service invocation tool** such as XClient, which is part of MVC, to test the remote web services, using the test request response messages created earlier. This should act as test harnesses for the remote web services and can be tested independently from the actual web services. Additionally, this enables the applications that will be built using these web services to be run independently, whilst exercising all the developed functionality as if the application was actually calling the real target web services. This can also be useful during application demonstrations, where the web services are not ready or inaccessible.

SCREEN DESIGN

- **Create a Skin**. This will provide a template for the screen, showing header, body, footer information as well as the menus, etc. Essentially the skin identifies a list of information groups and these groups/containers will hold different types of information on the screen. The forms group is the main container that will hold forms information for each screen. The skin is in XSL format and the appropriate groups are 'overridden' by each page during runtime. This ensures that all common elements of a screen are produced only once and the page-specific information is placed within each page.

- **Create a CSS**. The CSS should provide the appropriate styles for all information that is likely to be present on the screen. The CSS should also control the layout of the information groups defined in the skin. For example, the exact placement of header, body, footer and menus. This should also consider the placement and style of error information
- **Create an application navigation diagram**. This will show all screens and how they interact.

PROJECT STRUCTURE

At this stage it is important to dissect the overall application into separate projects to provide more modular development, source control and future maintainability.

- **Setup users, projects and configuration management**. At this stage, once individual projects have been identified, users can be defined that will be responsible for the projects. Please note that these users are not the same as the users in the source control system. For example an MVC user may have several projects assigned, but each project may be modified by several source control users. However, if the projects are aligned with MVC users and users defined within the source control system then this should minimise the need for merging of master project information. Please see *'MVC source control'* document for more information.

Please note that most of the design steps are independent of the MVC technology and demonstrates good-practice in XML-based application design.

USING MVC

FORMMAKER

Once the design elements are complete you can start to input and use this information in MVC.

- In the application map screen, **input the skin and CSS information** defined earlier. For each screen **indentify the WSDL/Schema** that will be used as a basis for the creation of the screen structure. Also generate or assign the instance documents that will be used for binding. This can often reuse the information created during the message design stage with some suitable MVC wrappers. Please note that MVC uses a default mvc:eForm wrapper within the middle-tier to encapsulate information. This wrapper is removed before interactions are made with the remote web services. However the instance information created during message design can often be repurposed for binding purposes using an mvc:eForm wrapper. The developer has full control over the wrapper that is applied and how the message is restructured at each stage.
- For each screen, **create the structure** from the selected WSDL or screen template.
- **Fine tune** each page element as required using the Field Details screen.
- **Bind** the screen elements to the instance documents in the Bindings screen

Please see the FormMaker Developers Guide for more detailed information on these stages.

XDE AND RULEMAKER

- Import the WSDL and **generate web service proxies** for each of the remote web services defined in the WSDL files. This will generate proxy services, including SOAP calls adhering to the message definitions.

- Create and link each controller to the generated proxy service access nodes.

- For each controller, decide what the **primary purpose of the controller** is. This is likely to be either a display only controller or an update controller. The additional complexity is when a controller needs to call multiple services before composing a screen and also call multiples save services once information has been captured on a single screen. At this stage it is worth designing and categorising such controllers.

- In order to maintain clarity, **Workspaces** should be defined to hold clearly identifiable information in separate documents.

Please see the Message Design guidelines provided earlier and the *MVC tutorials* for more information in this area.

TYPICAL ISSUES

ARCHITECTURE AND MESSAGE DESIGN

- **Web Services interfaces not agreed**. This can lead to lengthy test cycles and rework due to changing interfaces.

- **Parallel development of the user interface and web services tiers** can cause issues during development and integration testing. This is due to changing web services interfaces and message structures, causing unpredictable results.

- **Lack of isolated testing of web services and their interfaces independently**. This can increase the amount of debugging and tracing required during development, making it difficult to identify the exact location of errors within the distributed architecture.

- **Lack of Clones and Proxies for independent, standalone testing**. This can lead to increased development and test cycles due to changing parts of the application and also delays due to problems with infrastructure access. With appropriate clones in place, the testing can be carried out with minimum dependency on the actual remote systems.

- **Issues encountered with test data**. Without proper test data, the user interface can produce errors and unpredictable and inconsistent results. This can be reduced by aligning the request and response messages to the actual WSDL/XSD specifications of the remote web services. If the Message Design element is undertaken correctly then the text file based test data should result as a by-product.

- **Message Design and related aspects considered late, causing rework**. The lack of Message Design (XML Information Modelling) can cause many issues during development. This is irrespective of the technology used to implement the design. The better the design, the more rapid the development and more maintainable the resulting system will become.

TEAM SETUP AND GROUP WORKING

- **Enterprise architecture knowledge**. Although MVC can be used to construct applications rapidly, an appreciation of the overall architecture and message flow can greatly aid developer productivity, ensuring correct binding and orchestration of information at the correct stages, saving rework.

- **Training and appreciation of key XML concepts such as Namespaces**. MVC is designed for the creation of native XML applications. Developers can build XML applications without traditional programming knowledge, but an appreciation of XML and related technologies is required. Without this knowledge it can be difficult to trace message flows and isolate data problems.

- **Knowledge sharing and knowledge transfer.** Within large development teams it is important to ensure knowledge is shared across team members. If a problem has been solved once then it should be possible to publicise and share this knowledge to build critical mass.

Without a proper startup phase, including basic training and application design, this problem can be compounded due to multiple developers encountering the same problems at the same time without the opportunity to share solutions.

- **Source Control**. XML rich applications that rely on numerous XML text files are better suited to source control under 'copy-modify-merge' based source control systems rather than 'lock-modify-unlock' systems. Please see *'MVC source control'* document for more information.
- **Remote Team working.** This can cause issues with bandwidth speed and basic source control functionality. This, coupled with basic functionality of some source control systems, can lead to long delays during check-in and check-out operations. Some source control systems are designed for use over the web, rather than having basic web clients. These should reduce some of the remote team working issues.
- **Initial setup of projects under the mvc evaluation user causes issues**. Inconsistent blueprint files, causing build issues, etc. Once a project has moved from evaluation and proof of concept stages, it is important to partition the application into distinct projects and assign these to distinct users, leading to more modular development and maintainability. This also assists by reducing the merging and contention issues during multi-user development.