

Web Application Development for the SOA Age – ‘Thinking in XML’

Enterprise Web 2.0 >>> FAST

White Paper – August 2007

Abstract

Whether you are building a complete SOA architecture or seeking to use SOA services to build Rich Internet Applications (RIA), you will encounter massive amounts of XML information, representing the ‘common thread’ for such applications regardless of implementation technology. The ability to compose, send, receive, publish and efficiently process native XML information is vital to achieving robust, performant and timely Enterprise Web 2.0 and SOA solutions. Traditional development methods and tools often do not capitalise on, or hide some of the most important XML information, leading to systems that are rigid, brittle and very expensive and time-consuming to maintain. This paper will detail some of the common and important aspects that should be considered for modern web applications and a model driven approach that can significantly accelerate development of Enterprise Web 2.0 and SOA applications.

Introduction – ‘Thinking in XML’

If you are adopting SOA Services (SOAP, REST, RSS, Atom, XML/HTTP, XML JavaBeans, etc.) then you will encounter XML documents. If you are adopting modern best practices for web development using XHTML, CSS, AJAX, etc., you will encounter semantically rich XML content. This paper will focus on the modelling of XML documents that flow through typical RIA for SOA applications. These documents are often omitted or hidden deep within object structures within enterprise applications. Given the large number of XML documents that are present in SOA applications, it makes sense to consider them in more detail. Traditional application architectures, whether client-server or web-based, have primarily been concerned with screen design and object and component design. However, SOA architectures are heavily dependent on document exchange, validation, transformation and publication. For example, the default method for calling web services requires request, response and fault message structures.

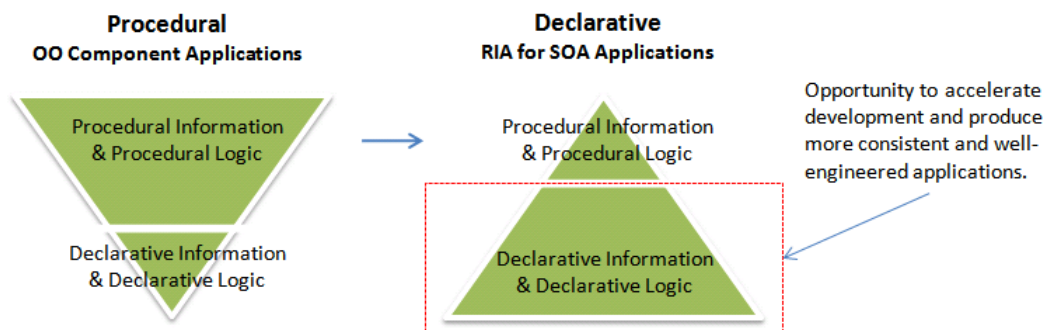


Figure 1. Declarative vs. Procedural applications from an XML Information Modelling perspective

Many approaches today rely on traditional 3GL coding using Java, C# and similar languages to implement the main business and orchestration logic for declarative applications. Based on the declarative service-oriented nature of such applications, the ability to model and declaratively process these documents presents significant opportunity to accelerate development and also produce repeatable and consistently well-engineered applications. In order to achieve this, a document modelling step is essential. This can:

- Reduce the ambiguity between Architects, Analysts and Developers
- Enable the construction of more agile and maintainable systems
- Reduce ambiguity between development teams working on different layers of the architecture
- Provide better separation between design and the underlying implementation technology
- Provide valuable documentation to enable easier future maintenance
- Accelerate development and reduce costs

In order to ‘tease out’ the relevant documents and the complexity they introduce, we will work through an example for the rest of this document. This example will introduce a traditional client-server based web application and evolve it to a set of SOA services and a RIA client. During this process the important XML documents will emerge, highlighting the need for the modelling of such documents within RIA for SOA applications.

Screens, Objects and Data – The Unbreakable Bond

A financial organisation holds its customer and product information within an in-house database. A pseudo object/data model for this part of the database may look like the following:

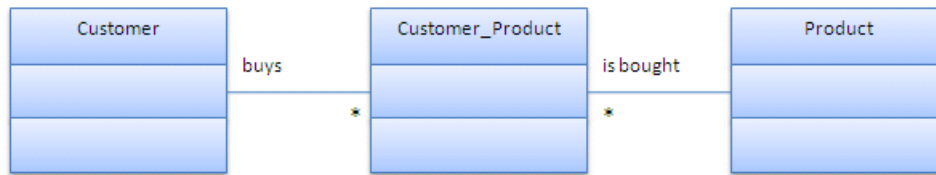


Figure 2. Pseudo 'Object' Model

The above model is used here purely to provide context for the example we are going to use to highlight some of the key XML assets that are present in RIA for SOA applications. The model has three objects/tables, **Customer**, **Product** and **Customer_Product**. The Customer object holds customer information; the Product object holds details of the products that are sold by the organisation. The Customer_Product object holds information about the products that have been purchased by customers. In the real world this database would almost certainly be physically partitioned between sales and administration, but let's proceed with a single high-level model for simplicity.

Various graphical interfaces are available to administer customer and product information as well as products bought by customers. Part of one of the sales-oriented interfaces may look like the following:

customer details

account number 12345678
forename John
surname Smith

product details

product code	name	purchase date	expiry date	initial amount	paid amount	arrears amount	balance
MORT0001	Mortgage	01-01-2001	31-12-2025	150000.00	45000.00	0.00	135000.00
LOAN0001	Personal Loan	01-01-2002	31-12-2007	10000.00	11000.00	0.00	2000.00
CARI0001	Car Insurance	01-01-2007	31-12-2007	500.00	250.00	0.00	250.00

Figure 3. Customer and Product Details Summary Screen

Within the screen, the customer details provide some basic information about the customer, together with a table that summarises the products that the customer has bought from this financial organisation. The image in Figure 4 shows what happens when the two main buttons shown on the screen are pressed.

When the **Edit Customer Details** button is pressed a separate AJAX-based partial page is 'opened' within the same screen to show additional customer details, together with the ability to amend and save the revised customer information. When the **Save Customer Details** button is pressed the modified information is 'saved' within the underlying data model.

When the **Buy More Products** button is pressed a list of available products is retrieved and displayed within a separate AJAX-based partial page, with the option to buy individual products.

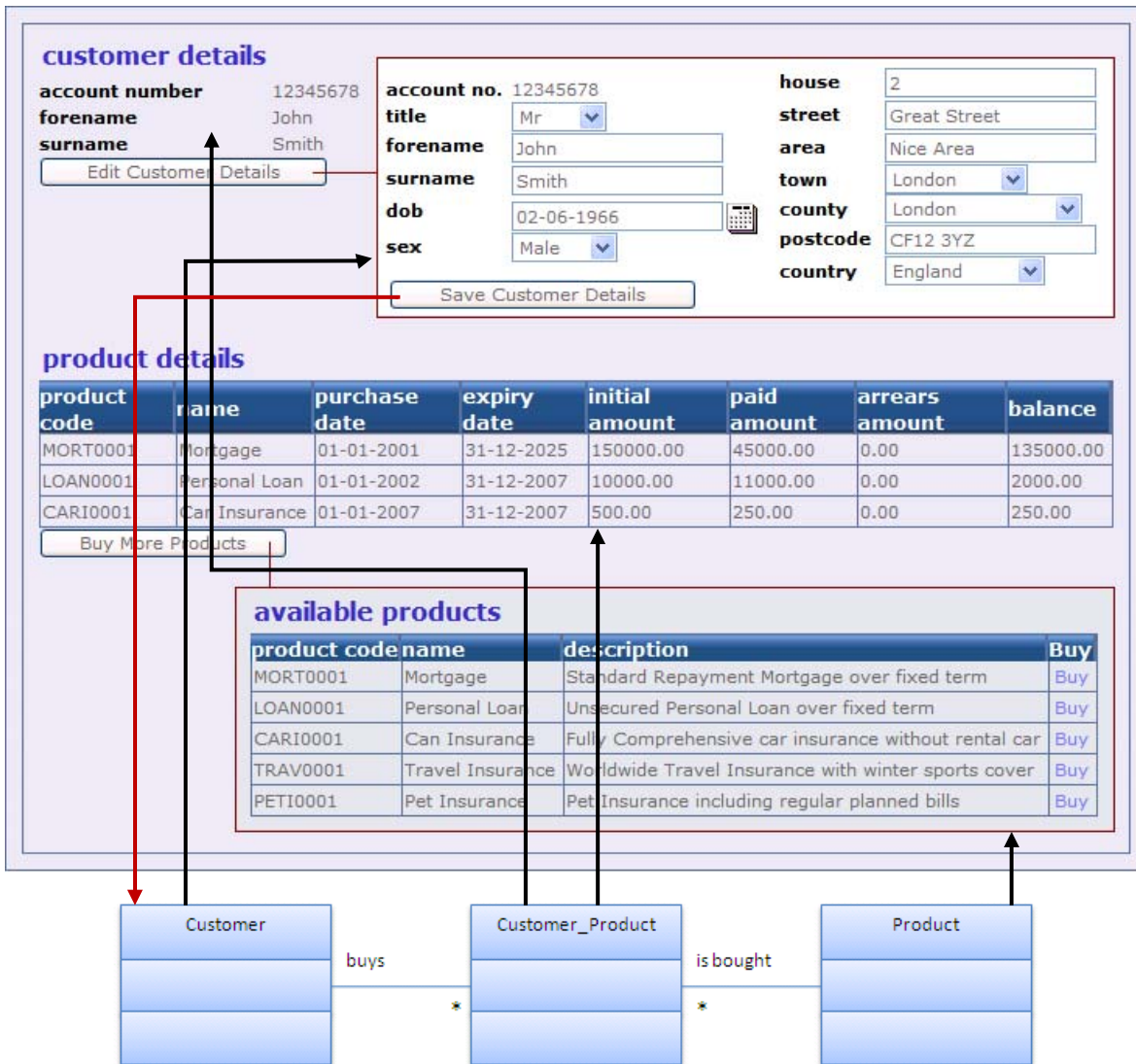


Figure 4. Detailed Customer and Product information with mappings to Data Model

It is probably not too difficult to visualise how this screen is constructed from the underlying object/data model, including the ‘save’ operations to update customer details. This example is not untypical of many client-server systems that are in existence today and the GUI may have originally been built using a client-server architecture. A web-based architecture would add an additional distribution layer, allowing remote web access but, without loose-coupling within a SOA architecture, the application would still have many direct mappings to the underlying information, creating a rigid structure that is difficult to distribute.

The SOA Part – Headless Services and Loose-Couplings

Now consider the following business requirement:

This financial organisation traditionally had a direct sales force that sold products to customers. A change in their business strategy means that the organisation will now primarily sell its products through financial intermediaries. The organisation will also provide a self-service web site for customers, wishing to purchase products directly. As a result the current sales support application needs to be deployed to third party organisations acting as sales channels. In order to meet this challenge, the organisation is considering redeveloping or adapting the existing system for the new business model.

The existing user interface is not distributable in its current form due to the dependency on the underlying data model, typically a standalone internal relational database. Significant investment has also been made in the existing application over a number of years and the cost of redevelopment from scratch is not an option due to massive costs and the need to implement a solution quickly. The company decides to reuse the existing server functionality. As part of the new architecture the organisation will provide a set of web services that can be used by different partner applications. Self-service websites for this organisation can also be built to utilise the same web services.

Figure 5 shows a possible set of web services, where the existing data model has been ‘wrapped’ to expose the core server-side functionality. The **Customer** service has operations including **getCustomerDetails** and **saveCustomerDetails** for retrieving and saving customer information respectively. The **Product** web service has operations including **getProductList** that can be used to obtain a list of Products that the organisation sells. There is also a **Customer_Product** web service that is intended to be hosted within a separate environment such as the Independent Financial Advisor (IFA) infrastructure or a separate architecture within other departments of the same financial organisation. This means that the data model is split between different physical locations and possibly technologies.

One of the main reasons for this split may be that the partner organisations that resell financial products hold basic customer and product purchase information, but the detailed customer and product information remains the property of the financial organisation supplying the financial products.

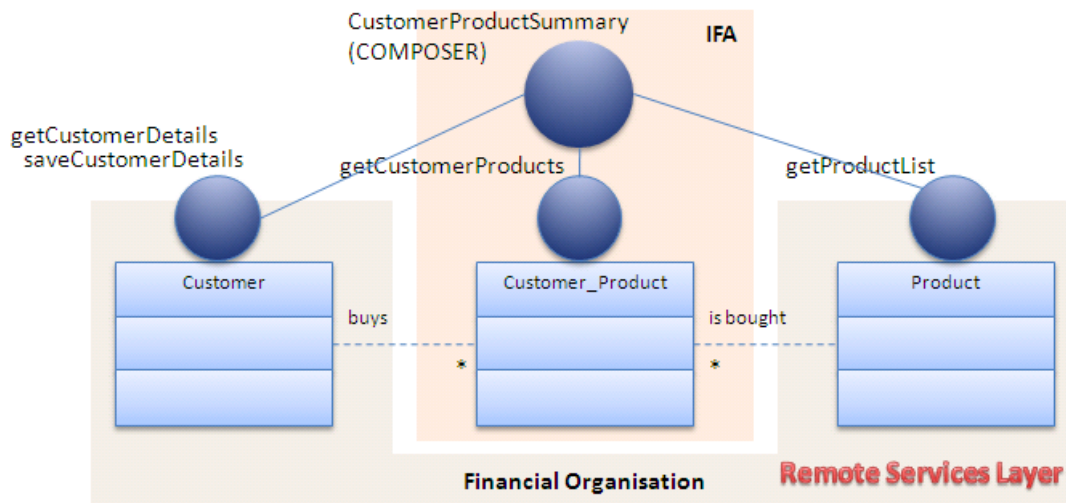


Figure 5. Web Services for a SOA Architecture

Technically, this means that the Customer_Product table resides within the IFA and the Customer and Product tables reside within the financial organisation. Since the Customer_Product information is only likely to have foreign references to Customer and Product, it is unlikely to be useful for constructing the Customer_Product Summary screen shown in Figure 4. To assist with this distribution issue, the organisation is providing a higher-level web service that will act as a **Composer**. The **CustomerProductSummary** web service will look-up the details of a specified customer within the local **Customer_Product** database and then make separate calls to the Customer and Product web services to obtain additional details for the Customer and Products.

XML Information Flows

Access to these services will be via ‘standard’ XML SOAP messaging. Figure 6 shows a template SOAP message structure is shown below. The exact structure and information within the Envelope will vary, but full details of this can be found within various standards available on the web. We will use the structure below to illustrate the messaging between the web services and their proxies that will use these messages to communicate with the web services.



Figure 6. SOAP message structure

Figure 7 shows an example schema of what the response from the CustomerProductSummary web service may look like. The response message contains details of customers and also the products that have been bought by customers.

The customer message structure shown in Figure 8 can be used within the response of the getCustomerDetails operation. The request for the operation is likely to need the customer account_number only. This same message structure could potentially be used for the request message of the saveCustomerDetails operation.

The possible structure for the products, returned by the Products web service, is shown in Figure 9. This message structure could be used to return the list of products. The request structure, if required, will probably only need to send any filtering conditions if the product list is likely to be very large.

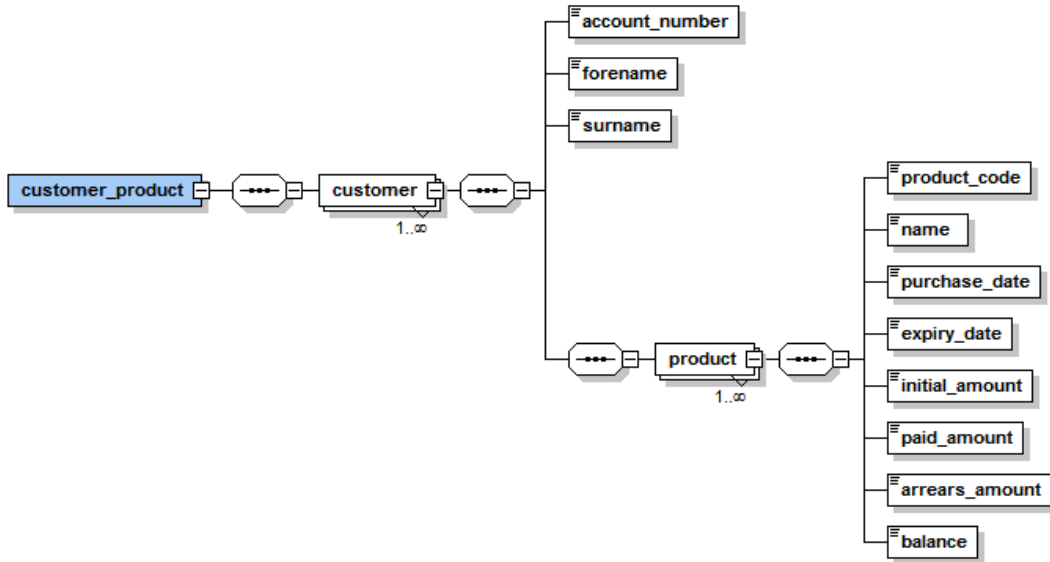


Figure 7. XML model for Customer and Product summary information

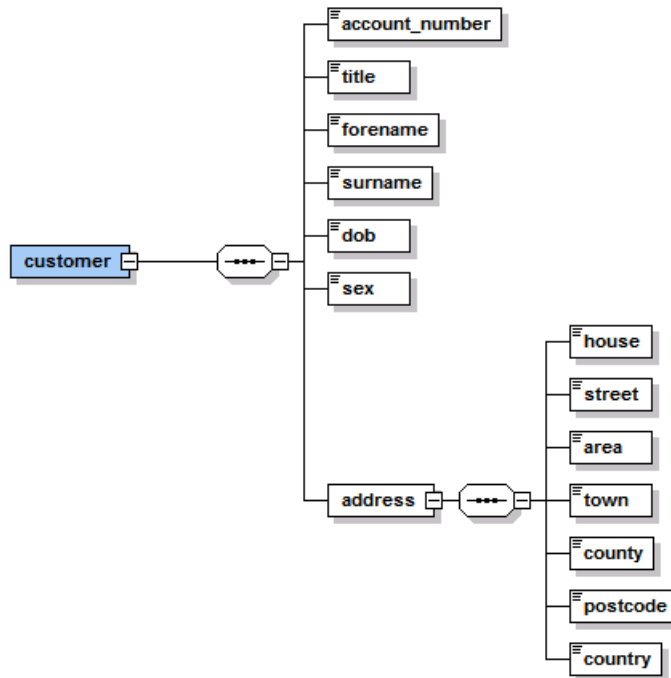


Figure 8. XML model for Customer details

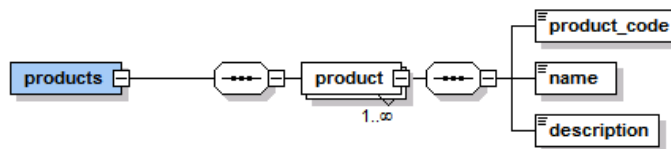


Figure 9. XML model for Product list

General schemas to interact with these services will include:

- Response Status Message structure to indicate the outcome of requests. This schema will typically be reused across all responses for a given application.
- Security Header Message structure. Again, this is likely to be consistent across services.
- WSDL for the **Customer** service
 - Request message schemas for **getCustomerDetails** operation
 - Response message schema for the **getCustomerDetails** operation
 - Request message schemas for **saveCustomerDetails** operation
 - Response message schema for the **saveCustomerDetails** operation
- WSDL for the **Product** service
 - Request message schemas for **getProductList** operation
 - Response message schema for the **getProductList** operation
- WSDL for the **CustomerProductSummary** service
 - Request message schemas for **CustomerProductSummary** operation
 - Response message schema for the **CustomerProductSummary** operation

The RIA Part – The Face of SOA

Now let us consider a typical financial advisor system, which resides within a separate organisation. This organisation has a partnership agreement that enables it to sell the products of the financial organisation mentioned so far. The financial advisor system may include a customer database, which includes basic customer information, together with a list of the products they have bought. To keep things simple let's assume that this IFA is a tied-agent and only resells products from this large financial organisation.

The objects/tables within the financial advisor system are limited. They hold the basic customer information, together with the products that customers have purchased. The CustomerProduct object only holds the 'link' information for the purchased products, with the details contained in the financial organisations systems.

The IFA has a similar interface to the UI originally present within the financial organisation. However, although the interface looks the same, this interface is a rich web application that is produced by composing information from remote services and managing this information within a SOA. A logical architecture of how this may be achieved is shown in Figure 10. This is very similar to the architecture shown in Figure 4, but using logical mappings to web services rather than underlying object or data models.

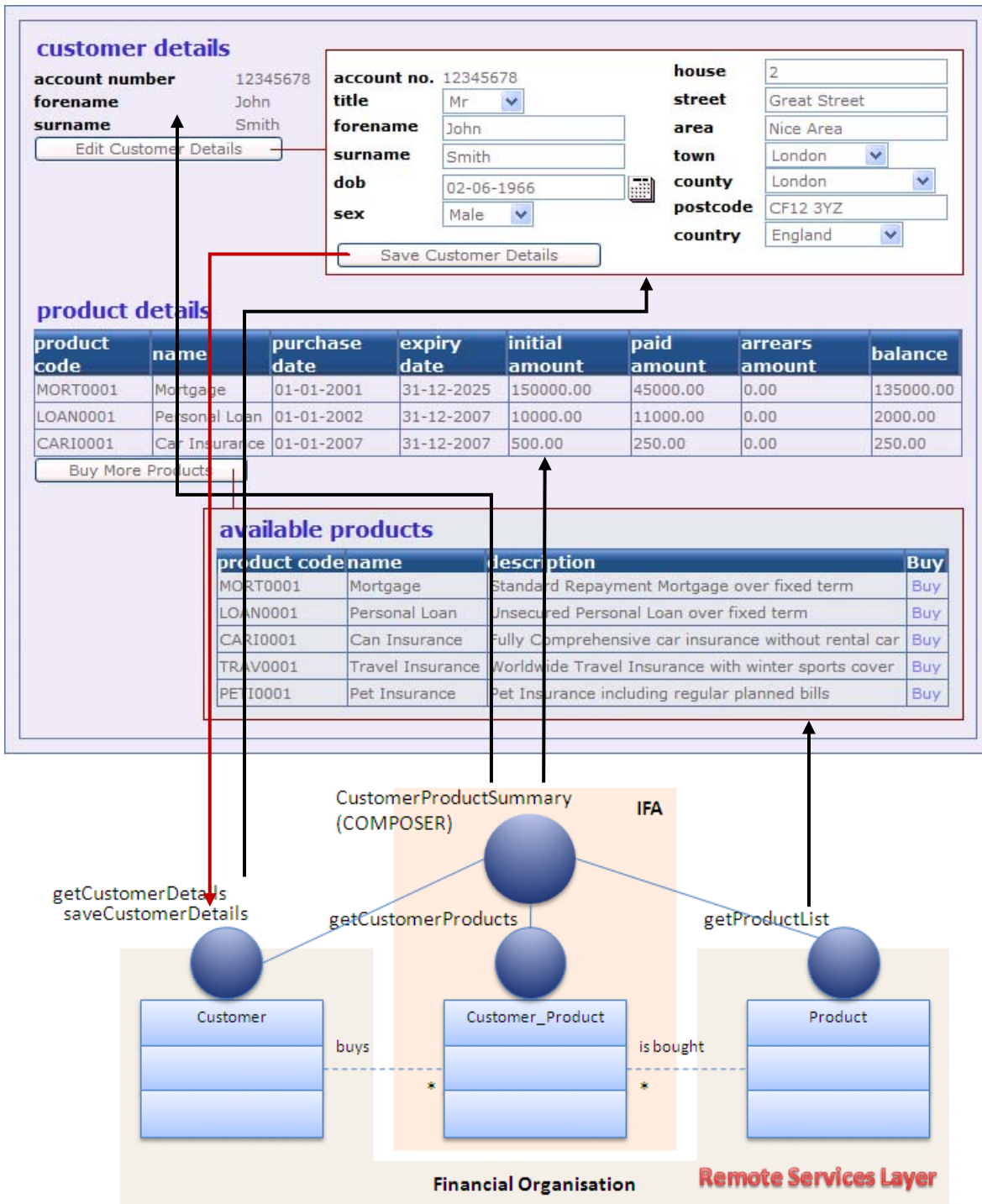


Figure 10. Logical view of RIA using the SOA Web Services

At a high-level this diagram may look ‘clean’ in logical form, very similar to Figure 4 in fact, with the objects/tables replaced by web services. However, due to the distribution of the loosely-coupled services, additional complexity is required in order to provide the same level of UI functionality as the original application.

Figure 11 shows a more detailed view of the architecture in order to enable the composition of remote services to produce the UI that is required. The layered architecture introduces separate dedicated layers for UI, Business Logic and Service Integration. The UI layer is focused on screen orchestration and mapping of information between the UI and the Business Logic layers. The Business Logic layer is focused on the Business Logic and uses the Service Integration layer to communicate with remote services. The right-hand side of the diagram shows the high-level message flows that occur within the architecture.

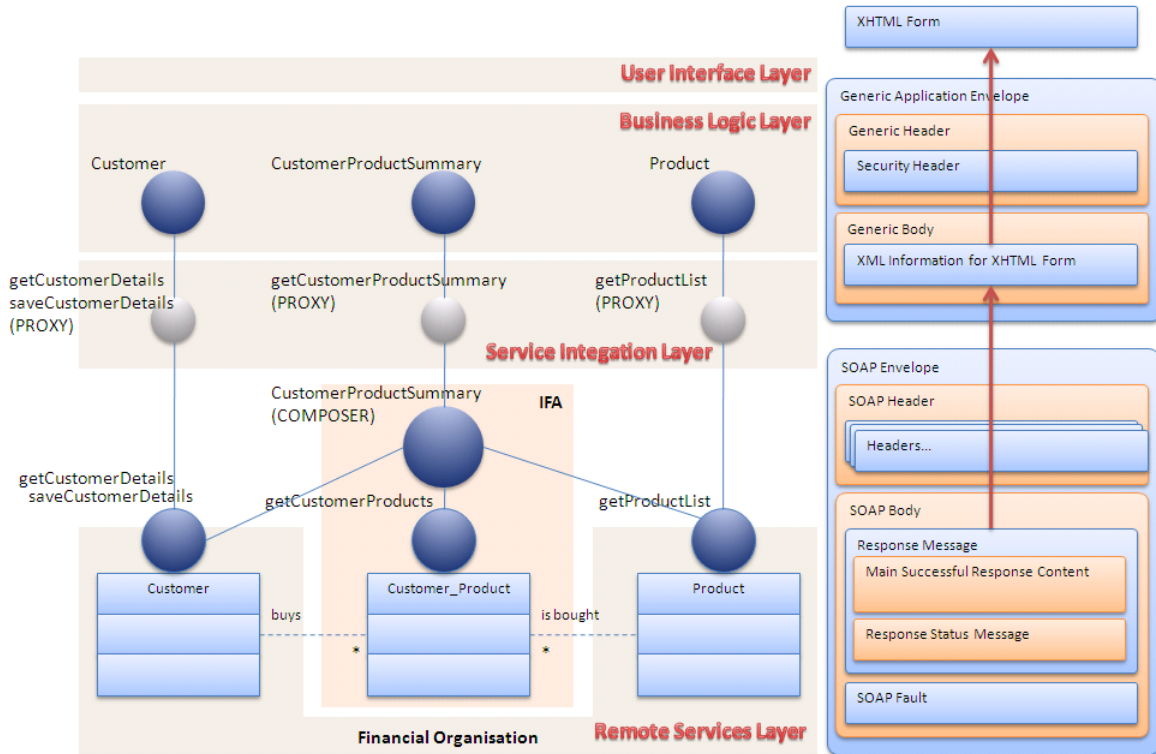


Figure 11. Creation of the Customer and Product Summary screen

XML Information Flows

Firstly you will notice that a *Service Integration Layer* is used to ‘encapsulate’ the ‘web service communications’ that are required to enable invocation of the remote web services and also handling of the response messages. This enables the *Business Logic Layer* to remain focused on the business logic rather than having to contend with low-level SOAP messaging.

Within this Service Integration Layer we need to consider the construction of the SOAP message structures before services are invoked. We need to consider the creation of the request messages themselves, together with the receipt of response messages and how these messages will be received and processed. We also need to consider the handling of any errors or SOAP Faults.

For Each Service operation we need to consider the following:

- Creation of the SOAP Envelope
- Creation of SOAP Headers, including Security Headers

- Creation of the request message within the SOAP Body
- Receipt of the Response Message status within the SOAP Response.
- Receipt of Errors and SOAP Faults.

After the communications level ‘plumbing functionality’ is completed the Business Logic Layer needs to receive the Response part of the SOAP Response and process as required. Typically, this response will not be in exactly the same format that is required by the UI and the mapping shown in Figure 11 between the Response Message and the XHTML information will not be easy. Due to this, further intermediate documents will result before the XHTML structure can be constructed for the UI. These intermediate documents often get lost within the design, but are essential both for development and longer-term maintenance of SOA applications.

Hidden Information Models to watch out for

Composite and Intermediate Documents

Creation of intermediate documents may be required in a composition scenario, for example, where the results of multiple service calls are aggregated to produce a composite document that is then used for UI creation.

An example message flow is shown in Figure 11. This may represent the SOAP Response returning the CustomerProductSummary information. The high-level message mappings are shown to the right of the diagram. In this scenario the composite service simplifies the document structure within the Business Logic Layer because the composition effort is provided by the CustomerProductSummary service.

Figure 12 shows a possible update operation such as saveCustomerDetails. This requires the creation of the XML documents and mappings between them in the opposite direction. A similar argument applies here in relation to the Business Logic Layer, related to the mapping between the information received from the UI and the Request message structure required by the remote services. There will typically be additional intermediate document structures required before the remote services can be invoked and the mapping shown between the XHTML message and the SOAP Request message will not be a single step. Again, these intermediate documents should be modelled to avoid information being lost during and after development and also to reduce ambiguity between design and development.

Hidden Information Models

The previous section is highlighting a typical issue that arises within many Business Logic Layers, that of ‘hidden information models’. What we mean by this is that as business logic is applied after receipt of SOAP responses or before the creation of SOAP Requests many intermediate documents are created. These may be fragments for drop-down lists, product pricing information, other database lookups, etc. Due to the prevalence of component-based logic in the Business Logic Layer these intermediate documents are often ‘hidden’ within method signatures and object attributes, making it difficult to follow the document flow.

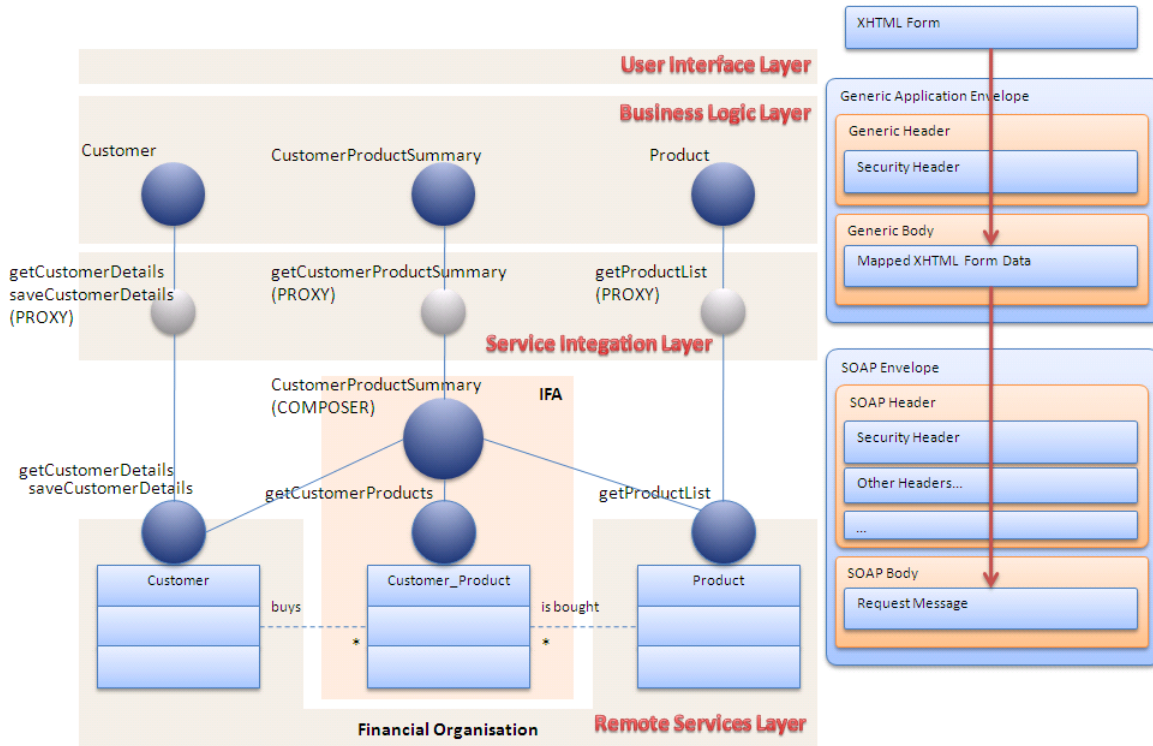


Figure 12. Saving modified Customer details

A more obvious example can be illustrated using the scenario in Figure 13. In this scenario the financial organisation does not provide the composite service that provides the CustomerProductSummary information. This means that the IFA organisation has to compose this summary information by making calls to the Customer, Product and Customer_Product services individually and aggregating the response information. The message flow to the right of Figure 13 shows that at least one composite document will have to be modelled to contain the responses from the separate service calls.

Error Handling

We briefly mentioned the Response Status messages within the SOAP Responses to indicate the outcome of remote service invocations. It is worth modelling this Response Status structure more formally to enable its capture after service invocations and also functionality to handle the responses. Based on the structure and content of the responses, certain orchestration decisions may be made to call other services or show particular screens based on the response data. This is also an area where error codes, descriptions, etc can be modelled for display and logging.

In this layer it is possible to compose the responses resulting from different service calls. This means that one has to compose a request message for each service call and handle the response following each service request. Each response will most likely strip out the soap wrapper since the response message is in the business domain by this point. Given that the response may be a successful response or a fault we could end up with many fault responses as well as successful response fragments. One point to note here is whether to capture all faults and display/handle each one after all calls have been made or make all calls but overwrite the fault fragment each time or stop after the first service call failure, which again may result in a single or multiple fault fragments depending on the error handling strategy that is adopted.

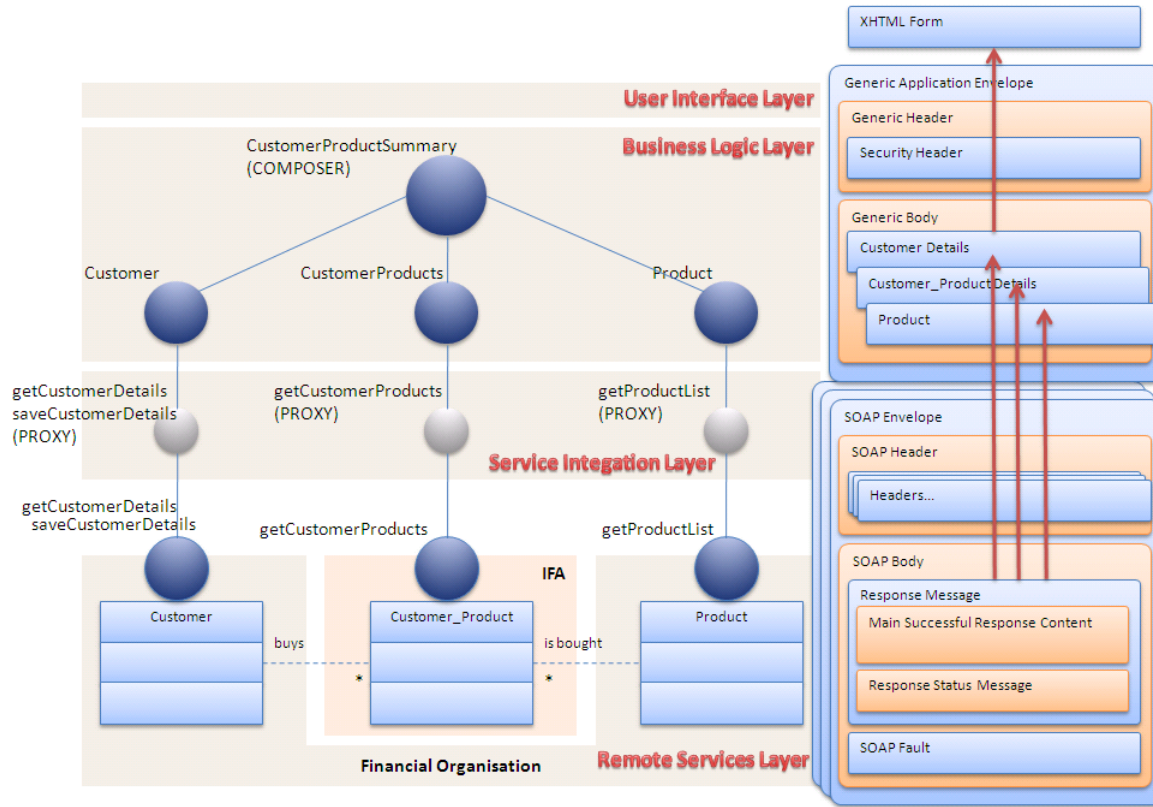


Figure 13. Creating screens using composition from different web services

Many applications will need to return messages that are not necessarily faults but indicators of transaction outcomes. For example, ‘successful’ message to indicate that a save operation has been successful or an error code to indicate a duplicate entry was attempted, etc. This could be intended application behaviour and is quite different to the ‘hard’ SOAP fault. It is therefore important to either model separate fragments within the composite document for such errors or have the capability to interrogate such messages, contained within the SOAP Body. Once again, the designer needs to decide whether to display all messages, the first error message or the last error message and also whether to stop after the first service has failed. This exact approach will be based on the dependencies between the different web services. For example, if the Customer Details Web Service is Unavailable (Returns a SOAP Fault) or the Customer does not exist (indicated by an error code in the web service response) then it does not make sense to call the Customer_Product web service since this requires a valid customer id to retrieve summary information.

The important point to note here is the difference between the types of faults and how they are handled and displayed. The reason this requires consideration is because the information model for holding the responses will vary depending on the approach that is adopted.

Security Headers

This information may typically follow industry standards for security information, but additional localised information may need to be modelled to enable role-based customisation of UI and controlling access to remote services and other application components. Customisation of the UI may control states for fields and groups of fields. For example, to control whether fields are hidden, shown, disabled, enabled, visible, editable, etc.

RIA to SOA Mappings

If a document-oriented declarative development approach is being adopted then it is worth considering the information mappings between the UI and the server. This typically means that the ‘flat’ form information within the UI is mapped to structured XML on the server. This preserves document flow and structure rather than mapping form information to objects, which can hide the information flow.

Cached Documents

Some information may require caching for later use either by the same session (such as session based security information) or application-wide use (such as the list of products or lists of drop-down values). The structures of these documents, which may result from remote service calls or loading of local configuration files, need to be modelled to ensure consistency when information is aggregated for future transactions. Again, rather than thinking about these documents as objects and attributes, it is often easier to visualise this information as documents because this is typically their native format.

The Business Logic Void

RIA for SOA applications are naturally document based due to the large number of XML assets that are present. Consider the diagram in Figure 14. Very often there is an implicit relationship between the remote services and the information being presented and captured by the UI. Also, the UI and the remote services are often document-centric due to their XHTML, HTTP, XML and SOAP foundations and the standards that are based around these technologies.

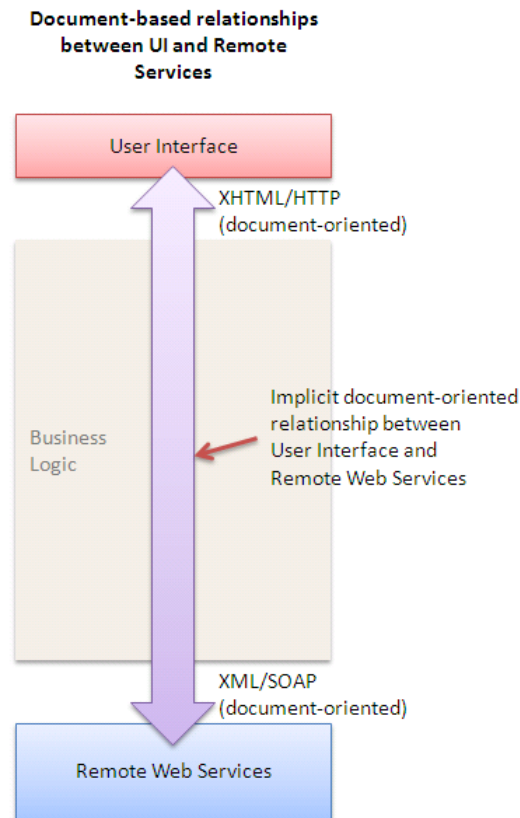


Figure 14. Document-based relationships between UI and Remote Services

In many implementations, old and new, the business logic layer in the middle is often component-based and acts as a ‘non-XML buffer’. This ‘hides’ and ‘interrupts’ the relationships between the different stages of the document flow. It is not to say that a component layer should not be used, but rather that the document flow should not be lost, even if this flow has to be preserved in design only. This can prevent issues between different development teams implementing different layers of the architecture and also issues arising during future maintenance of the system. If a 3GL approach is used, complete documents may end up as object attributes and many objects may result from a single document, which contribute to the ‘hidden’ information problem discussed so far. Within a document-oriented approach the flow of the documents and the changes in their ‘shape and structure’ as they flow between the UI and the remote services are preserved and ‘interesting’ from an information modelling context. See Figure 15.

It is not the intention of this paper to argue the case for or against either OO-based approaches or document-centric approaches, but the emergence of declarative logic engines does fit better with these types of RIA for SOA applications compared to traditional OO methods. From an information modelling perspective, whichever approach is adopted, it can be highly beneficial to model the intermediate and composite documents that are present within this large middle-tier.

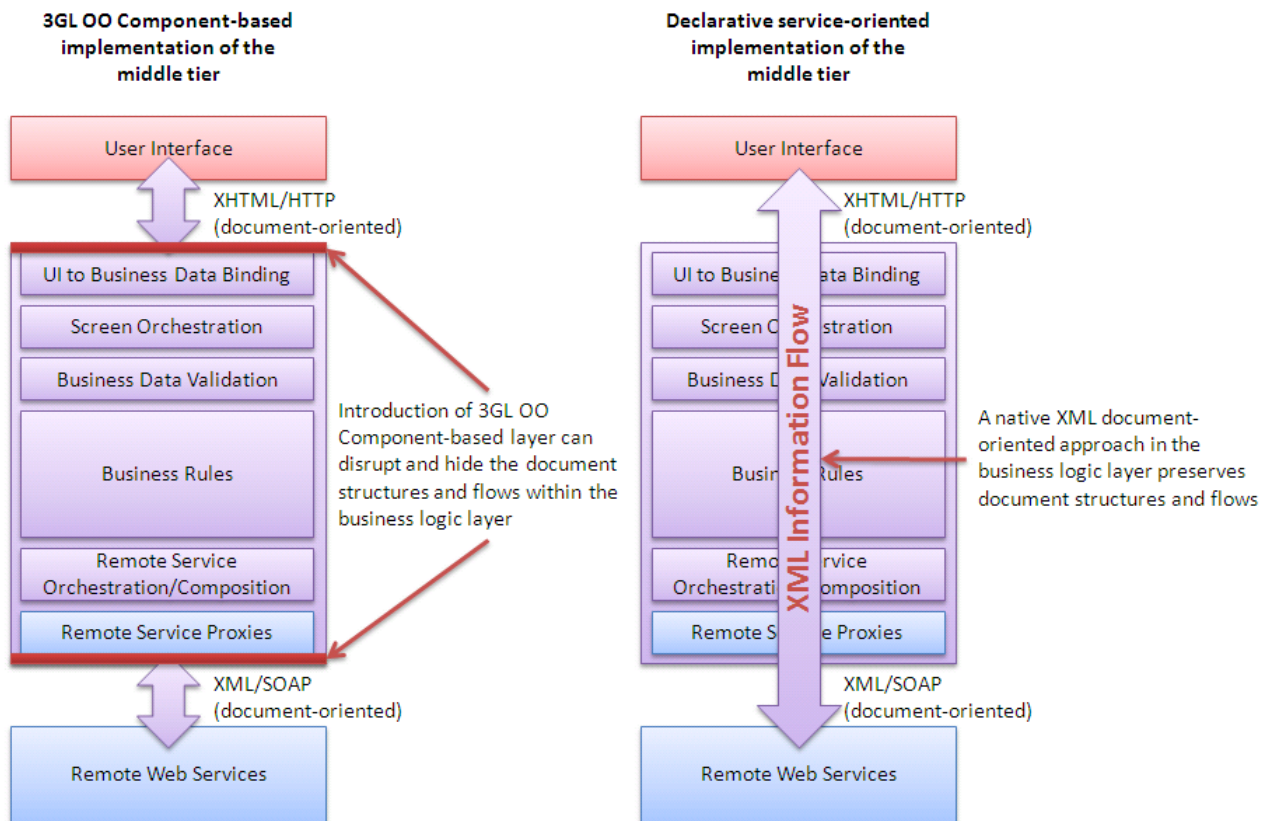


Figure 15. 3GL vs. Declarative implementation of the middle-tier (Preserving the XML Information Flow)

Conclusion

Enterprise applications are complex, even when all we are trying to achieve is a straight-forward replacement of existing client-server functionality. The new breed of flexible and composite Enterprise Web 2.0 and SOA applications are great, but that inevitable question remains. Is this web-based flexibility achievable without significant additional effort and associated risks, especially when we have to undertake significant additional XML modelling effort just to make sense of all the increased information flow within such systems?

The emergence of SOAs means we are forced to handle XML information regardless of the choice of technology for implementation. As shown in Figure 1, more of the application now consists of documents and these documents need careful consideration and handling to ensure we do not build systems that are cryptic and difficult to maintain. Although this may at first look like additional effort it is worth remembering that many of these documents will be freely available as part of web service contracts (WSDLs). Also, many additional XML documents will emerge as part of the modelling process. Therefore, systems and approaches that can capitalise on this rich set of documents will present significant opportunities to accelerate development .

It may surprise you to learn that there is a new generation of RIA for SOA technology platforms that can readily consume XML assets resulting from the information modelling stages detailed in this document to compose enterprise applications. It is possible to achieve this without the need for significant increase in development effort. However, this requires a document-centric approach (‘thinking in XML’) rather than an OO component based way of thinking. If a declarative, composite service-based approach is adopted and used correctly, it can provide better opportunities for automating many of the tedious development steps, resulting in systems that are faster to build, consistently better engineered and more maintainable.

The paper *‘Automating Rich Internet Application Development for Enterprise Web 2.0 and SOA’* may also prove valuable for this approach.

MVC – Rapid RIA Builder for SOA

MVC is a *Rapid* Rich Internet Application Builder for Service-Oriented Architectures. Unlike other tools, MVC has a complete design and deployment environment that can be used to build Enterprise Web 2.0 applications without any code, providing the *simplicity* of 4GL RAD environments. Our customers have built solutions, including:

- Legacy application modernisation
- High-touch transactional web applications
- Enterprise Mashups
- Contact Centre CRM systems and line of business systems integration
- Accessible Self-Service Websites

Affordable Quality

MVC is an *affordable* product and has provided savings of at least 60% during enterprise application development. MVC can be used to produce consistent and better engineered solutions that are standards-based and easier to maintain, with the agility to keep pace with rapidly changing business models.

Rapid automation through model-driven development

Using a model-driven development approach MVC can automate many of the mundane development tasks associated with traditional development tools and frameworks.

- Automatic generation of web service proxies (provides interactions with SOA services)
- Automatic generation of web pages
- Automatic data binding between XHTML and XML
- Automatic generation of validation logic
- Accelerated development through native XML and AJAX compatibility

Powered by Hyfinity’s SOA Machine

MVC is powered by Hyfinity’s SOA Machine, providing declarative, native XML rules-based orchestration and service composition capabilities. This can be used for data validation, screen orchestration, service orchestration, information composition and general-purpose business logic development, without the need for 3GL programming skills.

To learn more you can download a free copy of MVC and learn how you can develop applications in a matter of minutes for yourself. Please contact us at www.hyfinity.com if you wish to discuss your specific project requirements.

About Hyfinity

Hyfinity is a privately owned software company founded in 2001. Hyfinity’s mission is to simplify the development of Enterprise Web 2.0 applications by automating mundane developments tasks and reusing readily available information models. This is achieved through its ‘4GL RAD Style’ declarative service composition technology that can be used to build Web 2.0 applications without the need for 3GL programming skills.

Hyfinity has customers worldwide and is known for its innovation in rapid development tools for building Rich Internet Applications for Service-Oriented Architectures.

We work with our customers and partners to ensure solutions are designed and developed rapidly without compromising quality and scalability. We have worldwide OEM agreements with partners and some of our customer solutions include:

- Dynamic data-driven self-service web applications and eForms
- Feature rich, AJAX-based high transaction e-Commerce and internal solutions
- Contact Centres - CRM integration with existing line of business systems
- Service-oriented web application development for COTS Package Developers

Please contact us if you wish to discuss your specific project requirements or learn more about Hyfinity’s RIA for SOA technology and how it can help accelerate your next project development.



Hyfinity Limited
Innovation Centre, Central Boulevard
Blythe Valley Park, Solihull B90 8AJ, United Kingdom
(T) +44 (0)121 506 9111
(F) +44 (0)121 506 9112
(E) info@hyfinity.com
(W) www.hyfinity.com