2009 IEEE International Enterprise Distributed Object Computing Conference

Refactoring J2EE Application for JBI-based ESB: A Case Study

Wen Zhu, Walt Melo
Model Driven Solutions
Vienna, VA, USA
{wen-z,walt-m}@modeldriven.com

***Abstract*—Enterprise Service Bus (ESB) plays an important role in enterprise SOA, and Java Business Integration (JBI) is the standard for Java based ESBs. As enterprises adopt ESB as the integration hub, how to preserve and leverage existing J2EE assets becomes an important question. While most ESBs allow J2EE applications to be deployed as-is, refactoring these applications allows them to take advantage of many service provided by the ESB platform and prompts service reuse. We will discuss our experience with a major US federal agency's SOA efforts where a legacy J2EE application was refactored to be integrated in a JBI environment. We will also show how infrastructure concerns such Quality of Services (QoS) were moved from the J2EE application to the JBI container.**

Keywords-Service Oriented Architecture, Enterprise Service Bus, Model Driven Architecture, US Federal Agency

I. INTRODUCTION

Several authors have indicated that Service-Oriented Architecture (SOA) is fundamentally an architectural design approach: it aims at exposing enterprise business functionality and information as encapsulated services, and use these services whenever that functionality or information is needed [1]. Several mechanisms have been developed to make this architectural design approach feasible. One of these approaches is to provide an Enterprise Service Bus (ESB) as the message broker allowing service consumers and service providers to exchange messages in a SOA environment [2].

Despite the advantages of SOA and the benefits that an ESB can provide, IT organizations face several issues when trying to integrate their legacy application into a SOA environment. This paper discusses an approach which can be used to integrate legacy Java 2 Enterprise Edition (J2EE) applications into a SOA by leveraging the capabilities provided by an ESB. In this approach, legacy J2EE applications are re-factored and integrated into the ESB via service components. This paper also discusses a case study where this approach was applied. The case study was performed at a major US federal agency. A mission critical J2EE application was refactored and integrated into a standard-based ESB where quality of service (QoS) considerations were also addressed.

This paper is organized as follows. First, background information about SOA, ESB, and JBI is provided showing the main characteristics and benefits of these technologies and standards to this case study. Then, an overview of the environment where this case study was performed is provided. In addition, the non-functional requirements which drove the refactoring decisions are presented. The paper also discusses the approach used to refactor the legacy J2EE applications in the JBI environment. Finally, conclusions are presented.

We want to note that, while the technologies involved, including ESB and J2EE, are not specific to this project, we will focus on how to make the technology transition in a challenging business environment, where we applied software engineering principals including separation of concerns and Model Driven Architecture (MDA).

II. BACKGROUND

A. Service Oriented Infrastructure (SOI)

SOA is an architectural style for building software applications that uses services (typically web services) available within an enterprise or externally on a network, such as the Web [3]. SOA promotes loose coupling between services, allowing business applications to easily re-use existing services and resulting in applications that are easily and quickly built or changed. Service Oriented Architecture provides a level of flexibility that was not possible before in the sense that:

- A Service is a generic implementation of well-defined business functionality which can be used and reused by other applications. An important aspect of SOA is the separation of the service interface ("the what") from its implementation ("the how").
- Services are self-contained (perform pre-determined tasks) and loosely coupled (for independence).
- Services can be dynamically discovered.
- Composite services can be built by aggregating other services.

Service-Oriented Business Applications (SOBAs) are applications built on SOA principles. SOBAs run on top of a service-oriented infrastructure. A Service-Oriented Infrastructure (SOI) is an integrated set of software components that allows the creation, deployment, monitoring and management of enterprise services. This paper describes our experience with a specific SOI component – ESB.

B. Enterprise Service Bus (ESB)

An Enterprise Service Bus (ESB) is a flexible connectivity infrastructure for integrating applications and services [4]. An ESB powers a SOA by reducing the number, size, and complexity of interfaces between those applications and services. From an infrastructure perspective, an ESB addresses many of the enterprise application integration

1541-7719/09 $25.00 © 2009 IEEE
DOI 10.1109/EDOC.2009.25

IEEE computer society

requirements outlined in [5]. As we will demonstrate in this paper, an ESB provides many business benefits, including

- Facilitating reliable and seamless exchange of data among multiple applications (internal and external);
- Managing differences between multiple applications and business partners; and
- Providing an enterprise-wide, flexible, service-oriented approach to integration.

Our work was performed at a large US federal agency who acts as the purchasing agent for other federal agencies. From an organizational point of view, an ESB allows this agency to focus on its core business needs rather than the IT infrastructure required to connect the various programs. From an architectural perspective, an ESB allows the agency to add new services or make changes to existing services with little or no impact to the use of existing services, performing functions such as routing messages between services, converting transport protocols between requester and service, transforming message formats between requester and service, and handling business events from disparate sources.

C. Java Business Integration (JBI)

Introduced in 1995 through the Java Community Process (JCP), Java Business Integration (JBI) is the standard for Java-based ESBs [6]. JBI addresses the SOA integration needs by defining a standard meta-container, or a container of containers for the integrated services. Each JBI component is in turn a container for deployable artifacts, and messages are routed between services through a Normalized Message Router (NMR), modeled after the message exchange patterns (MEP) defined in the Web Service Definition Language (WSDL) 2.0 [7].

The JBI Specification defines two kinds of components:

- Service Engines (SE) that host business services implementing standard application programming interfaces (API). Examples of SEs include rules engines, Business Process Execution Language (BPEL) engines, and Enterprise Java Bean (EJB) containers.
- Binding Components (BC) that support external connectivity through standard communication protocols, such as Hyper-text Transfer Protocol (HTTP), Java Messaging Service (JMS), and Common Object Request Broker Architecture (CORBA).

An ESB based on JBI enables greater reuse and interoperability. For example, while all ESBs allow reuse of existing business implementations – applications developed with standard programming models – JBI takes reusability one step further by allowing ESB component reuse: components in one JBI ESB product (e.g. OpenESB) can be plugged into another JBI-based ESB (e.g. ServiceMix), thus avoiding not only lock in with a specific vendor, but also locking in with a particular open source community.

Furthermore, JBI specification encourages loose coupling by defining a standard Normalized Message Router where endpoints for services, hosted by SEs and BCs, are exposed and discovered. Through the use of binding components, JBI based ESB federation can be achieved through standard transport protocols such as HTTP.

As we will discuss later, the fact that services are exposed as WSDL endpoints greatly enhances the infrastructure maintainability. This is because the same metadata (WSDL) is used to describe both the ESB internal and external service interfaces, and policies can be specified using standard WSDL extensions, namely the mechanisms defined in the WS-Policy specifications [8].

There are many JBI-compliant ESB products, Service Mix and OpenESB being two of the most popular [9]. We should note that it is not the focus of this paper to provide an evaluation of various technology platforms and products, though product assessment and comparison were conducted as part of our work.

III. CASE STUDY

A. Environment

Like other large enterprises, the agency where our work was performed has used over the years different tools, techniques, methods, and infrastructure in its IT department. Business applications have been developed using a multitude of computer programming languages (e.g., COBOL, Algol, C, Java, etc.), and run on a variety of non-interoperable operating systems (e.g., ClearPath, Microsoft, Linux, Unix, etc.) and hardware. As such, it is under pressure to change and improve and face several challenges, including but not limited to:

- Difficulty to deal with changes. We live in a world where business practices, standards, federal policies, and technology are evolving and changing at a fast pace. However, the agency's current IT systems, due to their age, design, and architecture, make it difficult for the business line to easily respond to a changing and highly competitive market place.
- Considerable investment in legacy systems. IT legacy portfolios represent a significant investment for the agency. As such, it is important to leverage the current IT capital invested as much as possible. To achieve a modern IT infrastructure, this agency must adapt through innovation, while maintaining stability and leveraging as much as possible the capital invested in its current hardware and software infrastructure as well as its people.

The adoption of SOA and ESB can help this agency address the most pressing challenges and provide a stable, open and agile development environment and evolving architecture. Although SOA is not a panacea or the proverbial silver bullet that will solve all IT and business issues, SOA can, when implemented with proper planning and governance, go a long way towards overcoming many of these challenges. We will show in this paper how the use of standard-based service oriented infrastructure can help to address the challenges listed above.

B. As-Is J2EE Application Architecture

Many of the agency's legacy applications have been developed using J2EE technologies. As such, one of our

goals was to determine how to best leverage the existing J2EE assets in an ESB environment based on JBI.

Reusing existing J2EE business services are supported in a JBI environment through a service engine that is capable of hosting Java EE deployable artifacts, such as EJBs and web services. However, traditional applications are self contained. The infrastructure concerns such as external connectivity and security are handled by the application itself. Such coupling of technology and business concerns places severe limitations on the application's maintainability and flexibility. In such cases, refactoring a J2EE application is often a better choice since it allows the refactored application to take advantage of the ESB services while preserving the existing business logic implementation, resulting in a more modular composite application that is easier to maintain.

The example application used in our work to derive the refactoring patterns is a solicitation management system maintained by the agency. When a customer, also a US federal agency, wants to publish a solicitation through the system, it invokes the services through a web service interface that supports SOAP protocol over an HTTP transport [10]. The original application is a J2EE web application and the source code is annotated with J2EE web service annotations [11]. The as-is application architecture is illustrated in Figure 1.

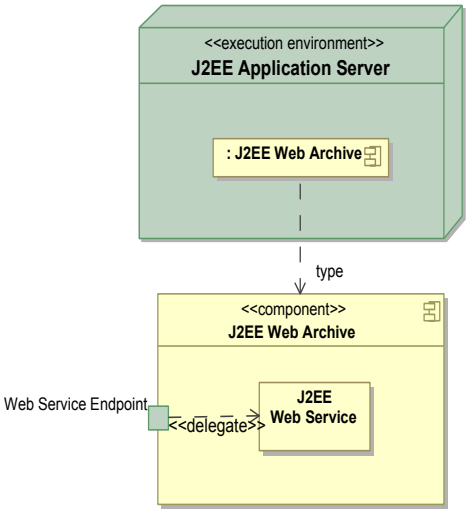

Figure 1. As-Is J2EE Application Architecture

C. *To-Be Architecture Requirements*

The key to refactoring an existing J2EE application to target an ESB platform is to first identify the infrastructure concerns that the application had to address; and then externalize these concerns to take advantage of the ESB platform. The resulting composite application should have infrastructure concerns addressed in different containers, clearly separated from the business logic.

A close examination of the application reveals many infrastructure concerns handled by the application as-is:

- Transport Binding: The application processes SOAP requests. Adding support for another standard transport protocol would not be a trivial task. In order to enhance the role of this agency as a service broker, it would necessary to provide further flexibility and increase the communication capabilities of the system under analysis. Clients might want to use SMTP, FTP, JMS, and other types of transport bindings to do business. It is impractical to impose a specific protocol.
- Authentication: The current implementation stores customer credentials in a database store. As a result, integration of an external authentication provider, such as an LDAP server, would require rewriting of the application code itself. It also limits the ability to take advantage of federated identity management or single sign on infrastructures. Therefore, it is important to separate the authentication services from the business logic. By doing so, the application could easily evolve and integrate with future infrastructure projects. For example, if the agency decides to use specific COTS for authentication, the business logic should not require modifications to conform to this change.
- Transaction Management: The current implementation relies on the database to manage transactions, making it hard to take advantage of the distributed transaction management infrastructures including WS-Transaction specifications [12] [13].
- Reliable Messaging: Often times an HTTP acknowledgement from a partner is not sufficient to confirm that a response has indeed reached the partner. One of the factors to consider is that a SOAP message may have to travel across multiple application layer intermediaries before it reaches the final application destination. Such intermediaries include gateways, protocol bridges and message adapters.
- Metadata and Policy Management: While services provided by this application are published in WSDL, Quality of Service (QoS) requirements are not explicitly defined. An example is the use of Transport Layer Security/Secure Socket Layer (TLS/SSL) mutual authentication. While this is a service interface requirement that a client needs to be aware of, it is not described in the service WSDL definitions.

Based on this analysis, we derived a set of non-functional requirements for the refactored application, including:

- Transport Binding: the refactored application shall take advantage of ESB's support for SOAP messaging over HTTP transport.
- Security: the refactored application shall rely on the ESB for application level security based on WS-Security specifications [14].
- Transaction Management: the refactored application shall delegate to the ESB the responsibility for transaction management.

- Reliable Messaging: the refactored application shall leverage ESB's support for the WS-ReliableMessaging specification to ensure reliable messaging [15].
- Metadata and Policy Management: the refactored application shall use WSDL based metadata with proper policy assertions.

It is worth noting that the refactored application supports the emerging SOA technology standards such as those illustrated in Figure 2.

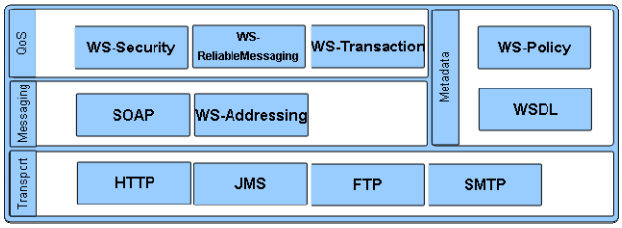

Figure 2. Infrastructure Standards

D. To-Be Application Architecture

The logical components used in the refactored application are illustrated in Figure 3.

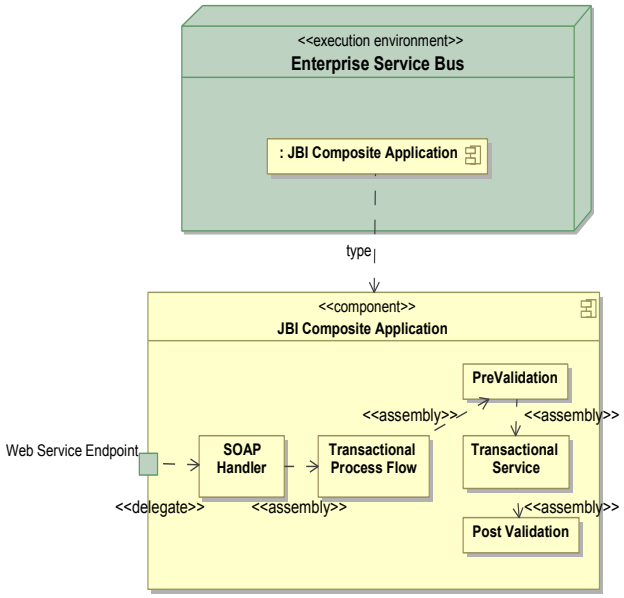

Figure 3. To-Be Application Architecture

The refactored application exposes the same functional interface as the original application. However, it is packaged as JBI service assembly (SA) with a set of JBI service units (SU), each to be deployed in a specialized JBI component which handles a particular aspect of the application functionality:

- SOAP/HTTP Binding. This service unit handles the logical service interface with the SOAP/HTTP protocol. The QoS requirements such as security, reliable messaging and web service transactions are also handled here.
- Transactional Process Flow and Validation Logic. A transactional process flow is introduced. While the process flow does not correspond to an original capability, it allows the introduction of business rule validations both before and after the invocating of the transactional service itself. It should be noted that the business rule validations are intended to ensure that pre- and post-conditions are satisfied. They are not for message schema validation which is handed in the SOAP/HTTP binding component.
- Transactional Service: The original business logic is deployed in the transactional service component, which ensures data integrity through the use of distributed transaction management protocols.

JBI services are exposed as WSDL endpoints on the Normalized Message Router (NMR). Accordingly, all the components in the refactored application implement the same WSDL interface (port type), with different binding types, and policy constraints expressed in WS-Policy assertions, attached to the WSDL elements.

E. Integration of Web Service Stack in an ESB

Java API for XML based web services (JAX-WS) is the Java standard for building web services [16]. In practice, a JAX-WS product, referred to as a web service stack, typically provides the following capabilities:

- Publishing a standard protocol-based web service endpoint;
- Hosting a web service implementation conforming to the JAX-WS specification. In most cases, the service implementations are Java classes with web service annotations;
- Enforcing web service QoS policies.

As such, a web service stack typically handles multiple infrastructure concerns for an application.

An ESB typically provides web service support by integrating a JAX-WS stack. In our cases, OpenESB is integrated with the Metro project, while ServiceMix relies on the Apache CXF. However, the fact that multiple infrastructure concerns are addressed by the stack causes different ESBs to take different approaches to integrate with the web service stack. Such integration approaches have important implications for the composite application design.

In OpenESB, the Metro web service stack is packaged in a binding component. As such, JAX-WS service implementations are typically deployed in a Java EE application server co-located with the ESB. The JAX-WS services are exposed on the NMR by a Java EE service engine. On the other hand, ServiceMix splits the Apache CXF into two JBI components – a binding component that provides the web service endpoint and QoS support, as well as a service engine that hosts the service implementation. Separating the protocol handling and service container responsibilities of a web service stack into two separate components provides a clear separation of concerns, which should result in better application maintainability. The drawback is greater complexity for the infrastructure.

IV. CONCLUSION

We believe that, by building on standard technologies such as JBI, an ESB platform achieves greater reusability and interoperability without being locked in with a specific vendor or specific open source community. At the same time, we have found that the metadata driven approach to policy management is adopted by the JBI-based ESBs. Such an approach in general, and the use of WS-Policy in particular, will improve the maintainability of infrastructure and applications, resulting in lower total cost of ownership.

Reusing existing assets, including logic implemented in J2EE technologies is one of the main objectives for refactoring a J2EE application for the ESB platform. Separation of concerns is a long-establishment principle in software engineering, and we believe that principle should guide the effort to transition legacy assets to an ESB platform. There have been many established practices to achieve separation of concerns, and a popular option is Aspect Oriented Programming (AOP). While AOP allows handling crosscutting concerns in an application, we believe a JBI-based ESB takes the separation of concerns practice one step further by allowing the externalization of infrastructure concerns out of the applications and delegating such the handling of such concerns to the runtime platform.

As we have demonstrated, existing J2EE code can be ported to ESB with little or no changes. Further, we believe Java EE application servers may have an important role to play in certain ESB composite applications when transactional supports are needed. However, whether an application server is appropriate for a particular ESB composite application should be decided based on a detailed analysis of the problem domain.

V. FUTURE WORKS

Many of the SOA technologies we used are still maturing. While we believe technologies like ESB and JBI are strategic fits for an enterprise SOA infrastructure, today's adopters will inevitably see many changes in the platform products they choose. As such, we are exploring Model Driven Architecture (MDA) as a way to further separate business concerns from technical implementation, thus helping enterprises preserve their IT investment in the face of rapid technology and product changes [17].

We are currently developing an MDA provisioning engine, ModelPro™, which generates deployable artifacts for particular platforms from Computational Independent Models (CIM) and Platform Independent Models (PIM). In particular, we are leveraging the Service Oriented Architecture Modeling Language (SoaML) standard from the Object Management Group (OMG). More information about this work can be found on www.modeldriven.org.

ACKNOWLEDGMENT

Many of our colleagues at Model Driven Solutions were involved in the project discussed here. We want to thank Cory Casanave for his leadership in the development of the SoaML specification and his vision for the ModelPro provisioning engine, as well as Ed Seidewitz for his leadership throughout the engagement. We also like to acknowledge Nilesh Kawane and Ted Tanaka for their contributions.

Our work could not have been possible without the support of our clients and partners in the US federal government. Especially, we want to thank George Thomas for his leadership on the Open Source eGov Reference Architecture (OSERA) initiative, which provided a foundation for much of our work.

Last but not the least; we want to thank the reviewers for their comments, which have helped us greatly in improving this paper.

REFERENCES

[1] Alex Cullen et al. "The Enterprise Architecture of SOA", Forrester, 2006.

[2] M. Keen et al. "Patterns: Implementing an SOA Using an Enterprise Service Bus", IBM, July 2004

[3] S Jones, "Toward an Acceptable Definition of Service", IEEE Software, vol. 22, no. 3, pp. 87-93, May/June 2005.

[4] M. Gilpin et al., "What is an enterprise service bus", Forrester, 2004.

[5] Fred A. Cummins, "Enterprise Integration: An Architecture for Enterprise Application and Systems Integration", OMG Press, 2002

[6] Java Business Integration (JBI) 1.0, JCP, August 2005, http://jcp.org/en/jsr/detail?id=208

[7] Web Services Description Language (WSDL) Version 2.0 Part 0: Primer, W3C Recommendation 26 June 2007, http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626.

[8] Web Services Policy 1.5 – Framework, W3C Proposed Recommendation, 06 July 2007 http://www.w3.org/TR/2007/PR-ws-policy-20070706.

[9] Larry Fulton, "The Forrester Wave: Enterprise Service Buses, Q1 2009", Forrester, January 2009.

[10] SOAP Version 1.2 Part 0: Primer (Second Edition) W3C Recommendation 27 April 2007, http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

[11] Web Services Metadata for the Java Platform. JCP, August 2003, http://jcp.org/en/jsr/detail?id=181

[12] Web Services Atomic Transaction (WS-AtomicTransaction) Version 1.1, OASIS Standard incorporating Approved Errata, 12 July 2007, http://docs.oasis-open.org/ws-tx/wstx-wsat-1.1-spec-errata-os.pdf

[13] Web Services Business Activity (WS-BusinessActivity) Version 1.1, OASIS Standard incorporating Approved Errata, 12 July 2007, http://docs.oasis-open.org/ws-tx/wstx-wsba-1.1-spec-errata-os.pdf

[14] Web Services Security SOAP Message Security 1.1 (WS-Security 2004), OASIS Standard Specification, 1 February 2006, http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf

[15] Web Services Reliable Messaging TC WS-Reliability 1.1 OASIS Standard, 15 November 2004, http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability-1.1-spec-os.pdf

[16] Java API for XML-based Web Services (JAX-WS) 2.1, May 2007, http://jcp.org/en/jsr/detail?id=224

[17] Bézivin, J, Gérard, S, Muller, P-, and L, R., "MDA components: Challenges and Opportunities", In: Metamodelling for MDA. 2003, York, England.