



Celtix Enterprise

An Introduction to Celtix Enterprise

Version 1.0
December 2006

Making Software Work Together™

An Introduction to Celtix Enterprise

IONA Technologies

Version 1.0

Published December 4, 2006

Copyright © 1999-2006 IONA Technologies PLC.

Trademark and Disclaimer Notice

IONA Technologies PLC and/or its subsidiaries may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this publication. Except as expressly provided in any written license agreement from IONA Technologies PLC, the furnishing of this publication does not give you any license to these patents, trademarks, copyrights, or other intellectual property. Any rights not expressly granted herein are reserved.

IONA, IONA Technologies, the IONA logos, Orbix, Artix, Making Software Work Together, Adaptive Runtime Technology, Orbacus, IONA University, and IONA XMLBus are trademarks or registered trademarks of IONA Technologies PLC and/or its subsidiaries.

Java and J2EE are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. CORBA is a trademark or registered trademark of the Object Management Group, Inc. in the United States and other countries. All other trademarks that appear herein are the property of their respective owners.

While the information in this publication is believed to be accurate, IONA Technologies PLC makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IONA shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Copyright Notice

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of IONA Technologies PLC. No third-party intellectual property right liability is assumed with respect to the use of the information contained herein. IONA Technologies PLC assumes no responsibility for errors or omissions contained in this publication. This publication and features described herein are subject to change without notice.

Table of Contents

Preface	v
What is Covered in This Book	v
Who Should Read This Book	v
How to Use This Book	v
The Celtix Enterprise Library	v
Getting the Latest Version	vi
Searching the Celtix Enterprise Library	vi
Additional IONA Resources	vii
Open Source Project Resources	vii
Document Conventions	viii
1. What is Celtix Enterprise?	1
What is Service-Oriented Architecture?	1
What is an Enterprise Service Bus?	6
Celtix Enterprise Is	10
2. Celtix Enterprise's Components	13
Components Build an ESB	13
The Celtix Enterprise Runtime	16
The Celtix Advanced Service Engine	17
Containers	18
Supplementing the Celtix Enterprise Runtime	19
3. Uses Cases where Celtix Enterprise Shines	21
Developing Applications with JAX-WS	21
Using Reliable Message Delivery	22
Web Service Enabling Existing Applications	22
Developing Services Using JavaScript	23
Deploying Smart Endpoints	23
Developing Asynchronous Services	24
Index	25

List of Figures

1.1. Distributed Bank Application	2
1.2. Separate Billing Systems	4
1.3. Billing Systems in SOA	5
1.4. Billing System SOA with an ESB	7
1.5. Distributed Nature of an ESB	9
2.1. Celtix Enterprise Components	14
2.2. A Celtix Enterprise ESB	15

Preface

What is Covered in This Book

This book describes the basic concepts involved in service oriented architecture and describes how Celtix Enterprise provides a powerful and open solution to creating a service oriented architecture. It does this by laying out the basic concepts and technologies surrounding service orientation. It then shows how Celtix Enterprise's components implement those technologies. The book concludes by discussing some common use cases where Celtix Enterprise is helpful.

Who Should Read This Book

This book is intended for all users of Celtix Enterprise. It is expected that the audience is familiar with general distributed computing concepts. You should also be familiar with SOAP, HTTP, and XML.

How to Use This Book

This book has three chapters:

- Chapter 1, What is Celtix Enterprise? discusses the key concepts in service oriented architecture.
- Chapter 2, Celtix Enterprise's Components describes how the components of Celtix Enterprise create an ESB. It also discusses the features and advantages of each component.
- Chapter 3, Uses Cases where Celtix Enterprise Shines describes how Celtix Enterprise can be used to solve a number of integration and application development problems.

The Celtix Enterprise Library

The Celtix Enterprise documentation library consists of the following books:

- Installing the Celtix Enterprise Binary Distribution describes the prerequisites for installing Celtix Enterprise and the procedures for installing Celtix Enterprise from the binary distribution. This is the preferred installation procedure for most users.
- Installing Celtix Enterprise from the Source Distribution describes the prerequisites for installing Celtix Enterprise and the procedures for installing Celtix Enterprise from the source distribution. This is only recommended for advanced users.

- An Introduction to Celtix Enterprise describes the components that make up Celtix Enterprise and how the work together. It also describes many of the concepts and techniques involved in SOA.
- Getting Started with Celtix Enterprise describes how to get up and running using Celtix Enterprise using a detailed example of creating and deploying a service.
- Using Celtix Enterprise provides detailed information on using Celtix Enterprise to develop and deploy Java services.
- Celtix Enterprise Command Reference is a quick reference to the commands you need when developing and deploying services with Celtix Enterprise.

The Celtix Enterprise GUI tools also include on-line help. To access it select Help → Help Contents. The help for the Celtix Enterprise GUI tools is in the section entitled SOA Tools Platform Developer Guide.

In addition to the above books, you may also want to read the documentation for each of the components that Celtix Enterprise bundles. This documentation is available from the projects responsible for developing the component.

Getting the Latest Version

The latest updates to the Celtix Enterprise documentation can be found at <http://www.iona.com/support/docs>.

Compare the version dates on the web page for your product version with the date printed on the copyright page of the PDF edition of the book you are reading.

Searching the Celtix Enterprise Library

You can search the online documentation by using the Search box at the top right of the documentation home page:

<http://www.iona.com/support/docs>

To search a particular library version, browse to the required index page, and use the Search box at the top right, for example:

<http://www.iona.com/support/docs/celtix/1.0>

You can also search within the PDF versions of each book. To search within a PDF version of a book, in Adobe Acrobat, select Edit → Find, and enter your search text.

Additional IONA Resources

The IONA Knowledge Base [http://www.iona.com/support/knowledge_base/index.xml] (http://www.iona.com/support/knowledge_base/index.xml) contains helpful articles written by IONA experts about Inferno and other products.

The IONA Update Center [<http://www.iona.com/support/updates/index.xml>] (<http://www.iona.com/support/updates/index.xml>) contains the latest releases and patches for IONA products.

If you need help with this or any other IONA product, go to IONA Online Support [<http://www.iona.com/support/index.xml>] (<http://www.iona.com/support/index.xml>).

Comments, corrections, and suggestions on IONA documentation can be sent to [<docs-support@iona.com>](mailto:docs-support@iona.com).

Open Source Project Resources

Apache Incubator CXF

Web site: <http://incubator.apache.org/cxf/>

User's list: [<cxf-user@incubator.apache.org>](mailto:cxf-user@incubator.apache.org)

Apache Incubator Qpid

Web site: <http://incubator.apache.org/qpid/>

User's list: [<qpid-user@incubator.apache.org>](mailto:qpid-user@incubator.apache.org)

Apache Tomcat

Web site: <http://tomcat.apache.org/>

User's list: [<users@tomcat.apache.org>](mailto:users@tomcat.apache.org)

ActiveMQ

Web site: <http://www.activemq.org/site/home.html>

User's list: <activemq-users@geronimo.apache.org>

Apache Incubator ServiceMix

Web site: <http://servicemix.org/site/home.html>

User's list: <servicemix-users@geronimo.apache.org>

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

<i>fixed width</i>	Fixed width (Courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the <code>IT_Bus::AnyType</code> class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: <pre>#include <stdio.h></pre>
<i>Fixed width italic</i>	Fixed width italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: <pre>% cd /users/<i>YourUserName</i></pre>
<i>Italic</i>	Italic words in normal text represent <i>emphasis</i> and introduce <i>new terms</i> .
Bold	Bold words in normal text represent graphical user interface components such as menu commands and dialog boxes. For example: the User Preferences dialog.

Keying conventions






This book uses the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, the command prompt is not shown.
-----------	--

%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the MS-DOS or Windows command prompt.
. . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
	In format and syntax descriptions, a vertical bar separates items in a list of choices enclosed in { } (braces).

Admonition conventions

This book uses the following conventions for admonitions:

	Notes display information that may be useful, but not critical.
	Tips provide hints about completing a task or using a tool. They may also provide information about workarounds to possible problems.
	Important notes display information that is critical to the task at hand.
	Cautions display information about likely errors that can be encountered. These errors are unlikely to cause damage to your data or your systems.
	Warnings display information about errors that may cause damage to your systems. Possible damage from these errors include system failures and loss of data.

Chapter 1. What is Celtix Enterprise?

Summary

Celtix Enterprise combines best of breed open source components to provide you with all of the tools needed to build and deploy applications into a service-oriented architecture. A service-oriented architecture is a loosely-coupled, distributed architecture in which services make resources available to consumers in a standardized way. A key piece of technology used in enabling service orientation is an enterprise service bus. An ESB is the infrastructure that handles the delivery of messages between different middleware systems, and provides management, monitoring, and mediation services such as routing, service discovery, or transaction processing.

What is Service-Oriented Architecture?

Overview

One of the keys to understanding Celtix Enterprise is understanding service-oriented architecture. *Service-Oriented Architecture* (SOA) is the next generation of software development methodology. It takes the concepts behind procedure-oriented design and object-oriented design and moves the layer of abstraction one step further away from the implementation details of a piece of atomic functionality. In doing so, it encourages the design of applications using loosely coupled units of functionality.

It also builds on the concepts used to create distributed applications such as CORBA. Specifically, it uses an XML-based grammar for defining abstract interfaces. The interfaces define the messages passed between service and consumer using XML Schema. By using XML-based types, service definitions make no assumptions about how the service is implemented.

Evolution of reusability in application design

The fundamental ideas behind service orientation are not new. One of the longest standing goals of software design is reusability. To achieve this, software languages and software design paradigms have evolved that encourage the compartmentalization of functionality. Functionality is grouped together into small, reusable units that can be used independently of the application for which they were originally intended. This not only makes them reusable, but also increases the ease with which large applications can be updated because a change to one unit of functionality does not necessarily require changes to the whole application.

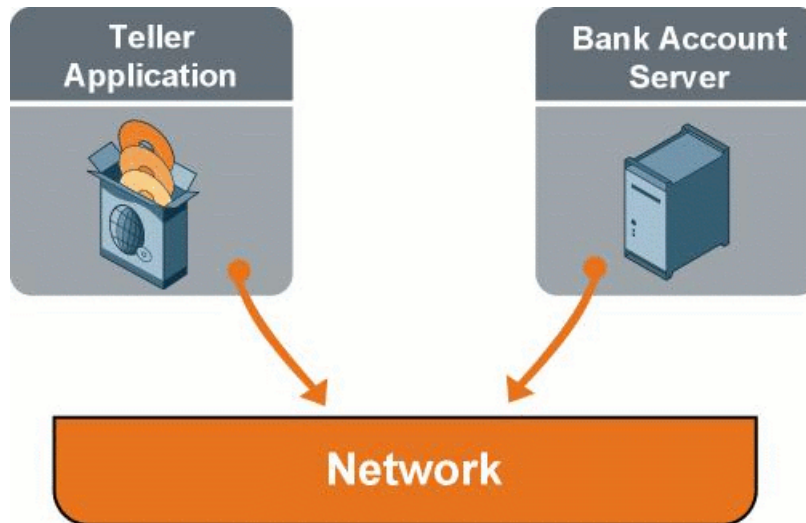
The first leap forward in the quest for reusability was the move from line-by-line programming languages like BASIC to procedural languages like Pascal and C. These procedural languages brought about the procedure-oriented design paradigm. Software began being designed as collections of reusable procedures each of which performed discreet pieces of functionality.

The next leap forward was the arrival of object-oriented programming languages like C++ and Java. Object-oriented languages, and object-oriented design, made re-usability easier by introducing the concept of an object as an atomic unit of functionality. In this paradigm, an object exposes a well-defined interface that can be called on by other objects that need the object's functionality. Because an object is a self-contained entity and because its interface is well-defined it is highly reusable across many applications.

The problems of distributed application development

As the code used to write applications became more modular and reusable, applications were being broken up into pieces that were distributed across many machines. For example, an application that allows bank tellers to make withdrawals and deposits is broken into a client and a server portion as shown in Figure 1.1, "Distributed Bank Application". The server portion may also be broken up into several separate parts.

Figure 1.1. Distributed Bank Application



Breaking applications into multiple parts and distributing them across multiple platforms presented a new set of reusability problems. Early distributed applications were designed so that all of the parts were tightly coupled. The messages used to communicate between them were passed using proprietary formats. Often there were dependencies on specific networking hardware and protocols. One result of this tight coupling is that pieces of functionality can not be reused because it is difficult to integrate these islands of functionality. For example, if a bank had two systems that needed to do credit checks, each system would need to implement that functionality because they used different messaging styles or different networking technologies.

Many attempts have been made to solve the reusability and integration problems posed by distributed application development. Some early solutions include CORBA, DCOM, MOMs, and large EAI servers. Each of these solutions got parts of the problem right, but never solved the entire problem. CORBA and many EAI solutions increased interoperability and reusability by providing abstract, implementation neutral definitions of atomic units of functionality that could be used as a contract between parts of a distributed application. MOMs increased interoperability by defining the interaction between parts of a distributed system by the messages that are exchanged.

None of the solutions really solved the problem because they, like object-oriented programming languages, did not provide a way of breaking the dependencies that bound all the parts together. CORBA required that all of the distributed objects be CORBA objects. EAI servers required resource heavy central hubs and proprietary networking solutions. MOMs required that all of the parts used a particular messaging infrastructure that often required specific APIs to be used.

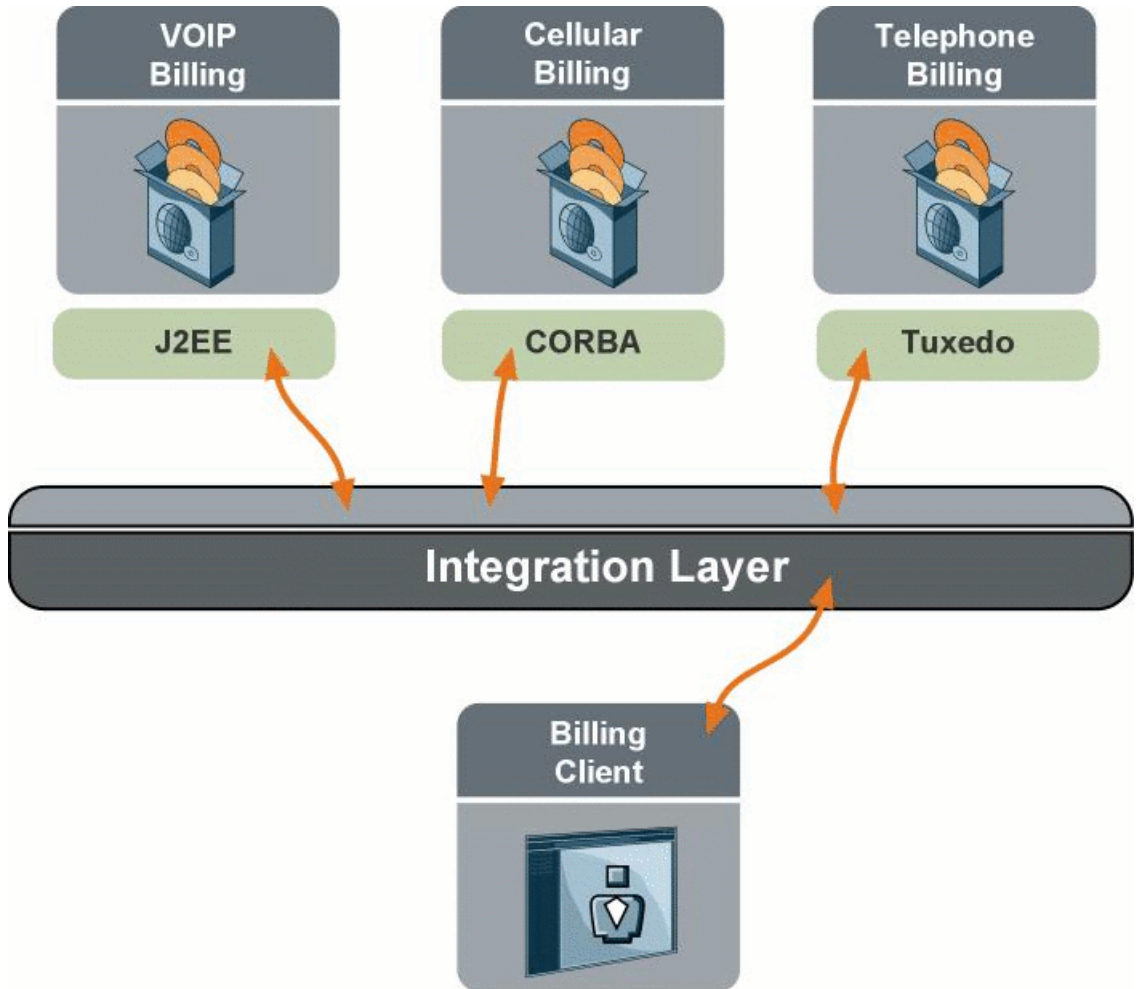
SOA breaks the dependency chain

SOA breaks the chains of dependency by borrowing from the best ideas of all other paradigms. From object-oriented programming, SOA borrows the idea of atomic units of functionality with a well defined interface. From CORBA and EAI solutions, SOA borrows the idea of an implementation neutral interface definition language. From MOM, SOA borrows the idea of defining applications by the messages they exchange. The result is the concept of a *service*.

A service is a unit of functionality defined by a set of message exchanges that are expressed using an implementation neutral grammar. A service, unlike an object, is an abstract entity whose implementation details are left largely ambiguous. The only implementation details spelled out are the messages the service exchanges. This ambiguity, coupled with the requirement that the messages be defined by an implementation neutral grammar make a service highly reusable and easy to integrate into a complex system.

Using services, you can define applications based on business requirements and not worry so much about the details of how the functionality is implemented. This is SOA. For example, you may need a unified application to generate customer billing for a telecommunications company that provides VoIP, cellular, and traditional phone services to its customers. The biggest stumbling block to this is that each department has implemented their billing system using a different technology as shown in Figure 1.2, “Separate Billing Systems”. Because none of the technologies were designed to be interoperable and none of them expose a common interface, building a unified billing client is a major integration headache. It can be solved using traditional means, but the solution involves either adding an expensive EAI product in the middle or developing a custom integration layer.

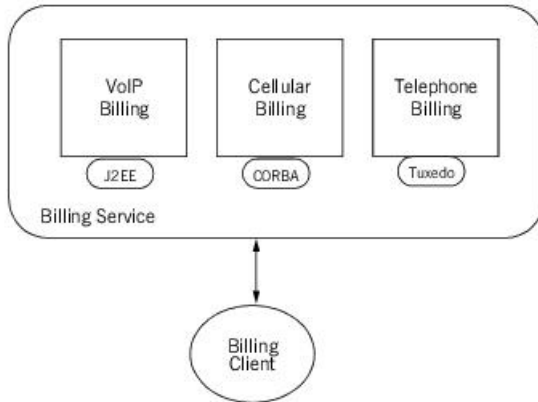
Figure 1.2. Separate Billing Systems



However, you can define a service that represents the functionality of all three billing systems as shown in Figure 1.3, “Billing Systems in SOA”. This service only requires one message exchange: the user sends the customer’s account number and the service returns the bill. You now have a common interface through which a unified billing client can access all three systems. This makes developing the client much simpler, will not require as much maintenance, and will make it easier to migrate the billing systems to newer platforms

if there is a business need. This approach is also much easier for a business level person to understand and express, thereby making it easier for an IT department to understand the requirements.

Figure 1.3. Billing Systems in SOA



Bringing a service into reality

The disconnect between SOA and real world applications is that a service is just an idealized representation of an implemented set of functionality. The implementation is still bound to the dependencies of hardware, languages, and networking protocols that go along with using computing resources. Several key technologies have emerged to bridge the gap between a service and the implemented functionality that it represents. Among these are XML and HTTP.

XML is the language that allows SOA to exist. It provides the grammar used to describe services, it provides the type system used to describe the data passed by services, and it provides the most common format used to package the messages used by services.

Web Service Definition Language (WSDL) is an XML grammar standardized by W3C to describe services. Using WSDL you define all of the abstract portions of a service including the elements that make up the messages exchanged by the service. You then map the abstract messages exchanged by the service to a concrete payload format that is used on a network. You also define a physical endpoint by which the service can be accessed.

XML Schema is the default type system for defining the messages used by a service. Because XML Schema is a standardized XML grammar it is platform neutral and does not make any assumptions about how the messages are going to be processed. It also allows for the creation of complex messages that are built up from reusable pieces.

Simple Object Access Protocol (SOAP) is an XML-based message protocol standardized by the W3C. It defines an XML envelope for wrapping messages and a data model for encoding information in an XML document. SOAP is the most common, but not the only, concrete message format used by services. Because it is XML based, SOAP is platform independent. In addition, it is widely used.

Hypertext Transfer Protocol (HTTP) is the most common network protocol used in SOA. This is largely due to the fact that it is nearly ubiquitous. HTTP is the protocol used to connect the World Wide Web and is based on an entirely open set of standards. Its ubiquity and openness make it a perfect backbone for connecting distributed services.

What is an Enterprise Service Bus?

Overview

An *enterprise service bus (ESB)* is the layer of technology that makes SOA possible. It creates the necessary abstractions by translating the messages defining a service into data that can be manipulated by the physical process implementing it. The ESB also provides the plumbing to connect the services to each other. In addition to the implementing the interface between an abstract service and a real-life implementation, an ESB also provides some of the qualities of service (QoS) such as security, reliability, and logging that are expected in enterprise environments.

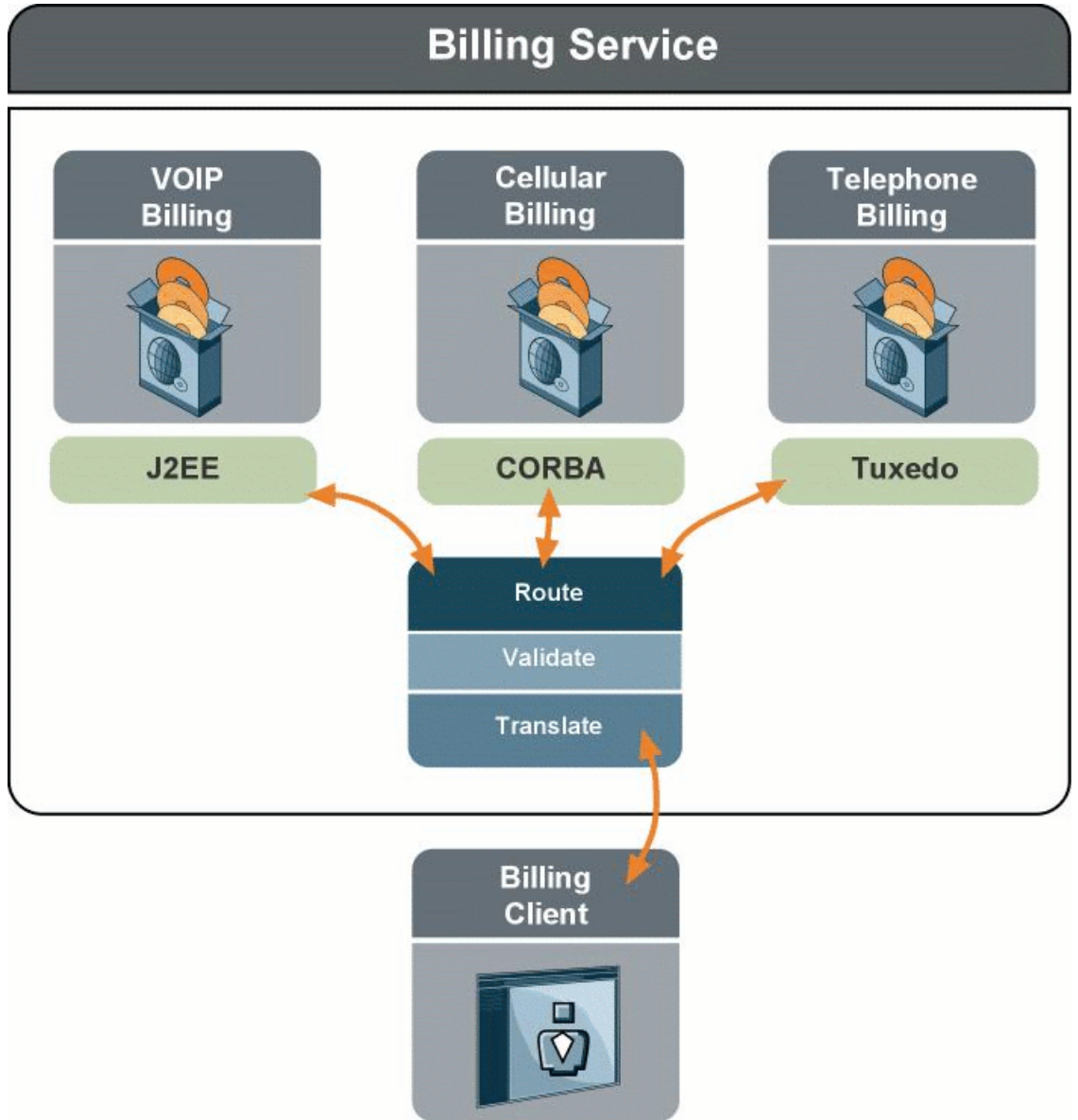
Connecting services to the world

An ESB takes the concrete details defined in the WSDL contract and uses it to make the service accessible to other resources on the network. This information includes details on how the abstract messages are mapped into data that can be manipulated and transmitted by the service's implementation. It also includes information about the transport and addressing details that other resources need to access the service. The *endpoint* is the physical representation of the abstract service defined in a WSDL contract.

As shown in Figure 1.4, "Billing System SOA with an ESB", the ESB sits between the service's implementation and any consumers that want to access the service. The ESB handles functions such as:

- publishing the endpoint's WSDL contract.
- translating the received messages into data the service's implementation can use.
- assuring that consumers have the required credentials to make requests on the service.
- directing the request to the appropriate implementation of the service.
- returning the response to the consumer.

Figure 1.4. Billing System SOA with an ESB



Not EAI

A brief description of an ESB may trigger nightmares about EAI. While the concern is warranted, ESBs have several key differences from past integration layers:

- ESBs use industry standard WSDL contracts to define the endpoints they connect.
- ESBs use XML as a native type system.
- ESBs are distributed in nature.
- ESBs do not require the use of proprietary infrastructure.
- ESBs do not require the use of proprietary adapters.
- ESBs implement QoS based on industry standard interfaces.

The use of standardized WSDL for the interface definition language and the use of XML as a native type system make an ESB future safe and flexible. As discussed in the previous section, both are platform and implementation neutral which avoids vendor lock-in.

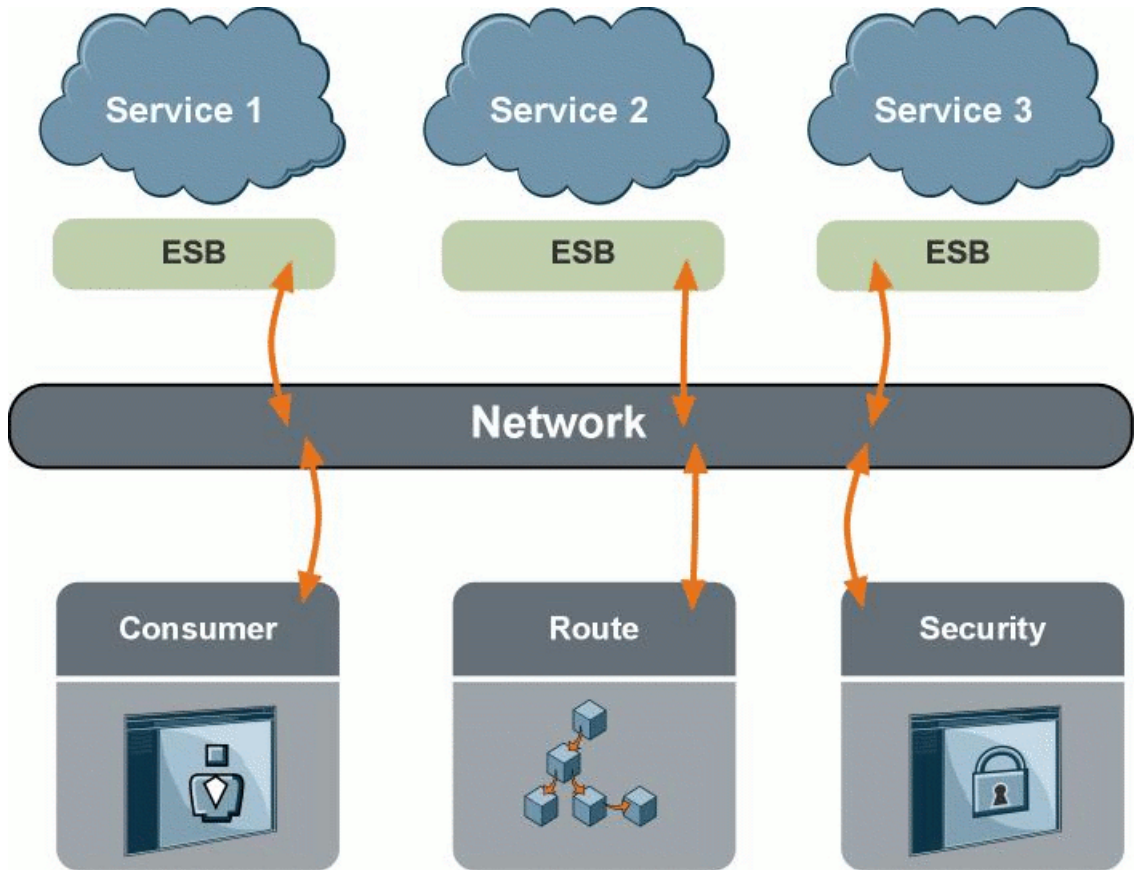
Strength in pieces

The key differentiators between ESBs and EAI systems are:

- ESB are intended to be distributed
- ESBs use standards based technologies

EAI systems were designed as a hub-and-spoke system. ESBs, on the other hand, are designed to be as distributed as the components they are integrating. As shown in Figure 1.5, “Distributed Nature of an ESB”, an ESB distributes the work of data translation, routing, and other QoS tasks to the endpoints themselves. Because the endpoints are only responsible for translating messages that are directed to them, they can be more efficient. It also means that they can adapt to new connectivity requirements without effecting other endpoints.

Figure 1.5. Distributed Nature of an ESB



EAI implementations relied on specialized adapters to connect applications together. There was no one standardized means for describing how the applications were wired together and different EAI implementations generally did not talk to each other. ESBs on the other hand use standardized technologies to connect applications together. The use of WSDL as the common contract language ensures that all ESBs understand how to expose a service. In addition, there are a number of standards that define how ESBs interact. Ideally, if you deploy multiple ESB implementations in your organization, they should all talk to each other.

The WS standards

ESBs offer a number of features such as transactionality, security, routing, and reliable messaging. To ensure maximum interoperability between implementations, ESBs base many of their features on a set of standards that include:

- WS-Addressing
- WS-Atomic Transactions
- WS-Coordination
- WS-Security
- WS-Reliable Messaging

These standards are all maintained by the W3C and provide a common framework on which ESBs build their functionality. They were all designed around the idea that information would be passed using SOAP/HTTP. They were also designed so that services could be easily shared and accessed over the Web. Therefore they, and ESBs that implement them are built to be maximally interoperable.

Celtix Enterprise Is ...

Overview

The short answer to "what is Celtix Enterprise?" is that Celtix Enterprise is an open source enterprise service bus (ESB). It is a collection of open source components that when used together supply you with the tools needed to build and deploy applications using service-oriented architecture (SOA). Celtix Enterprise builds on IONA's long experience in delivering enterprise strength distributed application frameworks, IONA's experience in developing and implementing standards, and IONA's recent experiences in the open-source community to provide a highly distributed and standards compliant ESB that is built using the best open source technology available.

History

IONA has a long history in the field of distributed software development. Fifteen years ago, we made our name as a supplier of a CORBA implementation. While it was not recognized at the time, CORBA was an early mechanism for developing applications using service-oriented design principles. The idea of finding a standardized mechanism for connecting reusable units of functionality was beginning to be solved.

In the last few years, IT organizations began to feel the pain of having much of their application resources locked up on remote islands an unable to communicate with each other. The promised reuse offered by CORBA and other solutions to distributed application design was not being achieved. To address these

concerns a new approach to designing distributed applications appeared. Service-oriented architecture provided a more loosely coupled approach to distributed application design that promoted reuse and being based on standardized technologies.

To address this need IONA created a highly distributed and extensible enterprise service bus called Artix. It was initially aimed at solving difficult integration problems. However, Artix was not limited to simply solving integration issues. Like IONA's CORBA solution, Artix was built around an extensible, distributed core so it also made it easy to incrementally move your applications to a SOA style design.

As customers began catching on to SOA, it became apparent that the core piece of infrastructure needed to implement and deploy SOA applications, the enterprise service bus, was an ideal piece of technology to be pushed into the open source community. Because it was central to the new approach to distributed application development, every IT organization was going to require one. Many IT organizations are mandated to use open source products when possible. Many do not have the budget to afford the licensing fees charged by large vendors. Many do not need all of the features of a commercial ESB.

To address this need IONA started the Celtix project in conjunction with Objectweb. The initial vision of the Celtix project was to develop a fully functional distributed enterprise service bus in the open source community. While the Celtix project was successful at creating an extensible framework for service development, it was not having as much success growing into a full ESB. It did not have a large enough community and there were a number of other projects all working on other parts of the SOA solution.

IONA decided that the best way to create a truly complete open-source ESB solution was to move the Celtix project into a larger community, narrow the project's scope, and team with other projects to build the remaining components.

The code and engineers from the Celtix project merged their efforts with the Xfire community to seed a new incubator project at the Apache Software Foundation called Apache Incubator CXF. This new project aims to create a powerful service development framework that provides the tools needed to implement services. Unlike a full ESB, it does not provide a deployment container or any messaging capabilities. It just provides the tools needed to plug a service into components that do provide that functionality.

To create Celtix Enterprise 1.0, IONA surveyed the open-source infrastructure landscape to determine which pieces would combine with Apache Incubator CXF to provide a first class ESB. The components that make up Celtix Enterprise include the Tomcat service container, ActiveMQ, and a router built using an open source routing framework. Together these components provide a truly extensible and flexible ESB to connect up your services.

Celtix Enterprise's approach to ESB implementation

IONA's heritage in distributed applications and that heritage shows in how Celtix Enterprise implements an ESB. Celtix Enterprise does not implement its ESB functionality using a central server or a specialized message broker. Instead, Celtix Enterprise embeds the ESB functionality into the endpoints implementing your services making them smart-endpoints.

Pushing the ESB functionality into smart-endpoints has several advantages:

- The smart-endpoints use resources more intelligently than a central broker.

Because they are independent, each smart-endpoint only consumes the resources it needs. They do not require the additional load to incurred to support the broker. In addition, because each smart-endpoint is independent, they can be dispersed over the available resources intelligently.

- It eliminates the potential bottleneck created by a central broker.

Smart-endpoints communicate with each other as peers. When a smart-endpoint creates a message, it has the intelligence required to package the data in a form that the recipient can read. When a smart-endpoint receives a message, it has the intelligence required to read the data and construct a message it can use. There is no need for a message to be shuttled through a central broker.

- The smart-endpoints are inherently flexible.

Because each smart-endpoint is independent they can easily be reconfigured to use different features without touching other parts of a production environment. If you have an application that you want to use reliable messaging, you simply need to reconfigure the endpoints involved in the application and redeploy them. None of the other parts of your environment need to be touched.

The other key to Celtix Enterprise is its pluggable architecture. This means that most pieces of functionality can be used independent of the others. For example, if you choose to deploy a smart-endpoint into a servlet container, the container will plug its own HTTP transport into the endpoint. A J2EE container may use its own JMS transport for connecting to a JMS broker. Celtix Enterprise's pluggable architecture also lets you add transports and bindings for custom messaging systems.

It is not just the deployment features that are pluggable. The method you use for developing your services is also pluggable. For example, you can choose to use the JAX-WS APIs to develop your services or you could choose to use Javascript. Celtix Enterprise also provides a limited plain Java API for creating services.

Why use open-source?

Open source provides a model for innovation that can produce high-quality features faster than traditional commercial models. It also reduces the dependence customers have on a single vendor, thereby reducing risk.

Integration and SOA is an ideal problem domain in which to collaborate using open source because it is a natural meeting point of many technologies created by many vendors. Open source lets all parties collaborate to make systems work together. It also removes the incentives to lock users into a particular technology or particular company's vision of how systems should interact.

Chapter 2. Celtix Enterprise's Components

Summary

Celtix Enterprise provides a set of components that fit together to provide all the functionality of an ESB. While each component is valuable, the Celtix Advanced Service Engine is the core on which the other components build. Based on Apache Incubator CXF, the Celtix Advanced Service Engine provides the runtime libraries that allow for the abstraction of Java objects to service interfaces. In addition to the service engine at its core, Celtix Enterprise includes a set of Eclipse based service development tools, a number of deployment containers, and two open source messaging platforms.

Components Build an ESB

Overview

As shown in Figure 2.1, “Celtix Enterprise Components”, Celtix Enterprise provides you a number of components that can be combined to provide a complete ESB solution. The core component in Celtix Enterprise is its service engine which is based on Apache Incubator CXF. You can chose the other components that wrap the service engine and to provide the additional ESB functionality you require. In each layer, you can select from any of the available options. You can also chose to leave out some of the layers such as the messaging layer or the intermediary layer. This pluggability makes Celtix Enterprise a customizable ESB solution that can be configured to meet your needs.

Figure 2.1. Celtix Enterprise Components

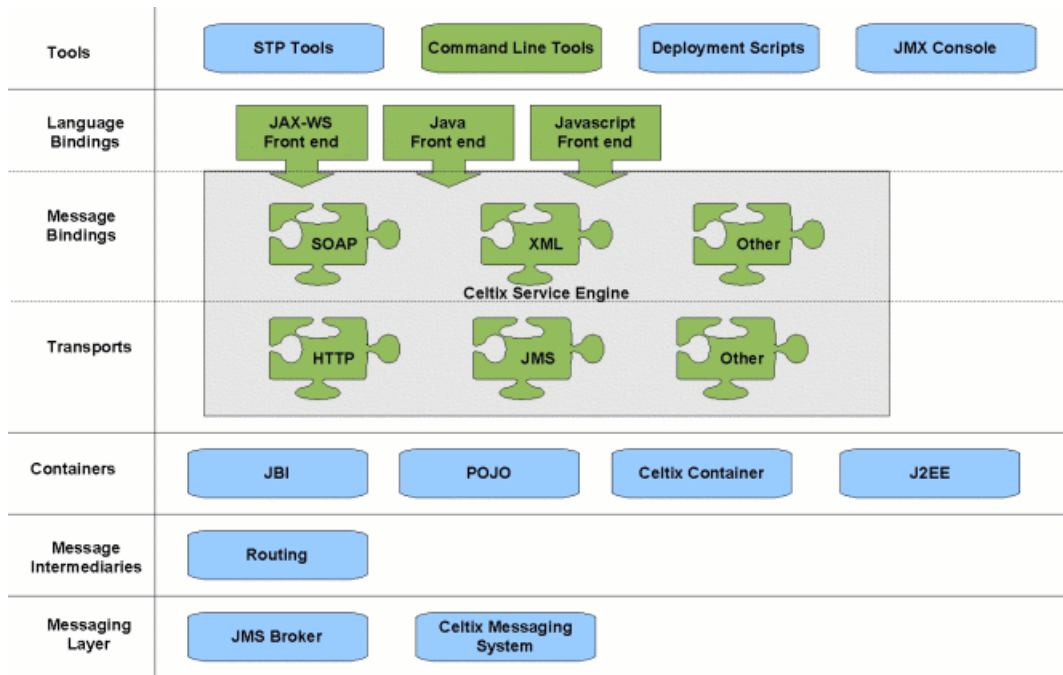
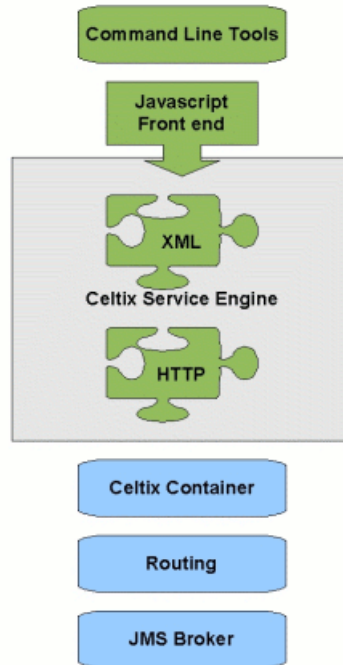


Figure 2.2, "A Celtix Enterprise ESB" shows an example of how you can combine the components to create an ESB. In this example, the service being implemented using JavaScript whose clients connect to it over JMS. The command line tools to are used to generate the starting point code. The JavaScript front-end provides the interface between the implementation and the layers that provide the connectivity. The service itself creates XML messages and connects to the network using HTTP. The router provides the bridge between the service's native message formats and the JMS broker. If you decided that your service did not need to be exposed over JMS you could remove both the router and the JMS broker. If you instead wanted to implement the service using JAX-WS APIs you could use the JAX-WS front-end.

Figure 2.2. A Celtix Enterprise ESB



Tools

Celtix Enterprise provides a suite of design time tools to aid in developing services. These tools are built using parts of the Eclipse SOA Tools Platform(STP). These tools provide a robust platform for service enabling existing Java applications, creating new services using JAX-WS, and creating the required metadata for deploying into a variety of runtime containers. In addition, there are a number of command line tools to generate code and WSDL that are based on Apache Incubator CXF.

In addition to providing design time tools, Celtix Enterprise makes it easy to monitor your services using one of a number of management consoles. Celtix Enterprise's runtime core is fully instrumented using JMX. Any JMX compliant management console will allow you to see metrics on the performance of services developed using Celtix Enterprise.

Celtix Advanced Service Engine

The core of Celtix Enterprise the Celtix Advanced Service Engine. Based on Apache Incubator CXF, the service engine contains all of the front-ends, message bindings, and transports that are used to implement a service. This component provides all of the hooks required to plug in different combinations of front-ends, bindings, and transports. It also provides hook to allow service to be plugged in to the lower layers of the ESB.

Containers

The service engine allows you to implement services and provides the libraries needed to connect the implementation to the network, but it does not provide an environment in which they can run. Celtix Enterprise provides you with three different runtime environments, or containers, for your services. In addition to the provided containers, Celtix Enterprise based services can be deployed into any J2EE container.

Intermediaries

Intermediaries provide additional functionality to the ESB by intercepting messages on the wire and doing some form of processing on the message before passing it along. Celtix Enterprise includes a routing intermediary that can direct messages based on different criteria, switch between different message formats and transports, perform basic transformations. You could also implement a costing service or a logging service as a message intermediary.

Messaging layer

Celtix Enterprise's messaging layer provides two messaging systems:

- ActiveMQ JMS
- The Celtix Advanced Messaging

Celtix Enterprise is designed to use any available messaging system and it provides most qualities of service regardless of what messaging system is used. The two advanced messaging systems bundled with Celtix Enterprise add one more layer of flexibility to how you build and deploy your services.

The Celtix Enterprise Runtime

The Celtix Enterprise runtime consists of three components:

- a service implementation
- the Celtix Advanced Service Engine

- a container

Together these components form a running service endpoint. The service implementation is the logic that performs the functionality defined by a service contract. The service engine provides the abstraction layer that connects the implementation to the outside world. The container manages the resources needed by the service implementation and the service engine.

The Celtix Advanced Service Engine

The Celtix Advanced Service Engine is based on Apache Incubator CXF. The service engine has a pluggable architecture that allows it to take advantage of several different programming models, several different message bindings, and several different transports. In addition to the basic features, the service engine also includes pluggable libraries to implement WS-RM reliable messaging and WS-Security based security.

Front-ends

The service engine provides front-ends for the following programming models:

- JAX-WS 2.0
- Plain Java
- JavaScript

Depending on the requirements of your project, you can choose to use any of the provided front-ends to develop a service.

The GUI tools provided with Celtix Enterprise work with the JAX-WS front-end. Celtix Enterprise provides a full implementation of the JAX-WS 2.0 specification for developing services. Using this front-end, you can annotate almost any Java object to create a service.

The other front-ends also make it easy to implement services. The JavaScript front-end is modeled after the JAX-WS front-end and is a quick way to get a service up and running. The Java front-end provides a limited set of methods to connect to the lower layers of the ESB and manipulate the data provided.



Note

The JavaScript front-end limits you to using the SOAP data binding.

Bindings

The Celtix Advanced Service Engine has a pluggable data binding interface. This pluggability allows you to change the message format used by an endpoint at deployment time. For example, you could deploy an endpoint for a bank services to exchange SOAP messages and you could deploy a second endpoint for the

same service that exchanged messages using native XML without the overhead of the SOAP envelope. You could also develop your own bindings to further expand Celtix Enterprise's capabilities.

In addition to the SOAP and XML bindings provided with Celtix Enterprise, you can also use Apache Incubator Yoko to CORBA enable an endpoint. Using a CORBA binding enables you to do the following:

- develop JAX-WS clients that interacts directly with CORBA back end services.
- develop services that interact natively with CORBA clients.
- incrementally migrate CORBA services and clients to a WS implementation.

Transports

The service engine has a pluggable transport interface. Using the pluggable transport interface, you can reconfigure the messaging layer used by your endpoints at deployment. For example, you could do all of your testing using HTTP and then deploy the production endpoint into an environment that uses JMS without changing the service implementation. You change the transport using either the WSDL or the endpoint's configuration.

Celtix Enterprise includes the following transports:

- HTTP
- HTTPS
- JMS

You can also develop and deploy custom transport plug-ins using the Apache Incubator CXF APIs.



Note

Some of the containers replace the Celtix Enterprise transports with their own transports.

Containers

Celtix Enterprise provides three runtime containers:

- the Apache Incubator ServiceMix JBI container
- the Tomcat 5.x servlet container
- a lightweight Spring based container

In addition to the provided containers, Celtix Enterprise supports the deployment of runtime components into J2EE containers such as JBoss.

Because Celtix Enterprise allows you to deploy your services, and consumers, into a variety of different containers, it allows you to use a familiar runtime environment. If your organization uses a J2EE environment you can develop services and deploy them that way. If you are interested in moving toward a smart endpoint strategy for service deployment, Celtix Enterprise's lightweight container is a perfect solution.

Supplementing the Celtix Enterprise Runtime

Overview

The pluggable transport interface used by the Celtix Advanced Service Engine frees Celtix Enterprise from being dependent on any specific messaging infrastructure. However, many users want to use a more robust messaging system than plain HTTP. To address this need, Celtix Enterprise includes two messaging systems that can be used in addition to supporting HTTP.

In addition to adding messaging systems to your service environment, you can also use Celtix Enterprise to add features such as routing, security, and orchestration. The components that implement these features are not a part of the Celtix Advanced Service Engine or one of the containers in which a service runs. They are implemented by intermediaries deployed into your network. The intermediaries interact with the other applications deployed as part of a service-based application.

Intermediaries

Celtix Enterprise provides a routing intermediary. It has the following features:

- transport switching
- binding switching
- content-based routing
- operation-based routing
- message transformation

You can also add orchestration and other advanced features to Celtix Enterprise using intermediaries. There are demos included that demonstrate adding a number of these features.

Messaging systems

Celtix Enterprise includes ActiveMQ JMS as a messaging system. Using the JMS broker you can capitalize on the qualities of service it offers. These include message logging, persistence, message prioritizing, and security among others.

In addition, Celtix Enterprise includes the Celtix Advanced Messaging. The Celtix Advanced Messaging is an AMQP implementation based on Apache Incubator Qpid. It offers a reliable and interoperable messaging system.

Chapter 3. Uses Cases where Celtix Enterprise Shines

Summary

Celtix Enterprise can help you address a number of integration and application development problems.

Developing Applications with JAX-WS

Overview

The Java API for XML Web Services (JAX-WS) is a vendor neutral specification for developing Web services using Java 5. It evolved from the Java API for XML-based RPC (JAX-RPC) specification and provides a number of benefits. Using JAX-WS prevents vendor lock in because it is a JCP standard. It also means your developers can work with a standardized, well-defined programming model.

How Celtix Enterprise helps

The GUI tools provided by the Eclipse SOA Tools Platform project provide JAX-WS compliant code from WSDL contracts and help you develop applications using it. The GUI tools also provide a mechanism for generating a JAX-WS service from a Java object. The tools help identify which classes can be exposed and guides you in adding the proper annotations.

In addition to the GUI tools, Celtix Enterprise includes the command line tools provided by the Celtix Advanced Service Engine. The command line tools generate JAX-WS code from a WSDL and assist you in adding bindings and endpoint information into your WSDL. There are also tools for generating WSDL from properly annotated Java objects and from XML Schema documents.

More information

For details on using the JAX-WS APIs see the following:

- Chapter 1, Developing a JAX-WS Consumer in *Using Celtix Enterprise*
- Chapter 2, Developing a JAX-WS Service in *Using Celtix Enterprise*

Using Reliable Message Delivery

Overview

In an enterprise environment it is often crucial that messages are delivered between endpoints. Many messaging systems provide reliable message delivery when using them. HTTP, however, does not have any built-in reliable messaging features.

How can Celtix Enterprise help

Celtix Enterprise provides two ways of developing services that use reliable message delivery:

- Use ActiveMQ, or another JMS implementation, as your messaging implementation.
- Use the WS-ReliableMessaging (WS-RM) feature of Apache Incubator CXF.

Using WS-ReliableMessaging

WS-ReliableMessaging (WS-RM) defines a mechanism for implementing reliable message delivery using SOAP/HTTP. Celtix Enterprise provides a full implementation of WS-RM. It is completely pluggable so that it can be deployed at runtime or not depending on the requirements. The Celtix Enterprise implementation uses a persistent message store to hold incomplete messages so that messages persist when one of the endpoints dies. When the endpoint reconnects, the messages will be delivered.

Web Service Enabling Existing Applications

Overview

You may need to expose the functionality of existing applications as Web services. While you may want to re-implement the existing applications using different technology, this is not always possible. It may not be cost effective to move all of the data to a different platform. Your deadlines may be too tight to do the redesign. In these cases you will need to find a way to expose the existing application as a Web service.

How Celtix Enterprise can help

If your existing application is implemented using Java, Celtix Enterprise's GUI can assist you in adding the needed JAX-WS annotations to your objects. It will then generate the WSDL which defines the newly created service. Once your Java code is annotated and the WSDL is generated, you can deploy the code as a service using any of the containers supported by Celtix Enterprise.

If you need to expose the functionality of a CORBA application, you can use Celtix Enterprise's router and Apache Incubator Yoko. The router, when configured to use Apache Incubator Yoko's CORBA binding, can act as an intermediary that transforms between CORBA messages and SOAP/HTTP messages. Using this approach does not require any coding. You simply need to create a contract that defines the functionality of the CORBA application you want to expose as a service, a SOAP binding that will be exposed to other endpoints, and a route between the CORBA endpoint and the SOAP endpoint. Once the router is deployed, the CORBA application will think it is talking to another CORBA application and the external Web services will think they are talking to a proper Web service.

Developing Services Using JavaScript

Overview

JavaScript, while many consider it a Web development language, is a powerful dynamic language. You can use it to develop services that are portable and lightweight. In most cases a service developed in JavaScript has significantly fewer lines of code and takes significantly less time to develop.

How Celtix Enterprise can help

Celtix Enterprise has a JavaScript front-end that allows you to write services and consumers using either JavaScript or E4X. The JavaScript front-end is modeled after the JAX-WS front-end. You implement the SEI in a JavaScript function. Celtix Enterprise also provides a runtime environment for your JavaScript services to run.

More information

For more information on developing applications using Celtix Enterprise's JavaScript front-end see Chapter 3, Using Dynamic Languages to Implement Services in *Using Celtix Enterprise*.

Deploying Smart Endpoints

Overview

One of the core principles behind Celtix Enterprise is the idea that services should be deployed as smart endpoints. A smart endpoint has all of the functionality to send and receive messages as specified by the service's contract. It does not rely on a central server and is more flexible in terms of resources and deployment options.

How Celtix Enterprise can help

The Celtix Advanced Service Engine is designed to create smart endpoints. Service's built with Celtix Enterprise can be deployed as standalone processes that are completely autonomous. You can also deploy them into the lightweight service container provided. Either mode of deployment provides you a lightweight smart endpoint.

More information

For more information on deploying smart endpoints see:

- An Introduction to Celtix Enterprise
- An Introduction to Celtix Enterprise

Developing Asynchronous Services

Overview

While synchronous request processing is appropriate for requests that take short amount of time to process, they may present an unacceptable bottleneck in situations when the requesting process can continue doing work while waiting for a response. In such cases it is more appropriate to use asynchronous messaging. There are a number of ways for applications to handle asynchronous messaging. Two common methods are callbacks or polling. In the callback scenario, either the service or the runtime calls a method on the requesting process using a special callback object. In the polling scenario, the requesting process periodically checks to see if any messages have been received.

How Celtix Enterprise can help

JAX-WS defines two standard mechanisms for implementing asynchronous consumers. The Celtix Advanced Service Engine provides the support to make these mechanisms work. In addition, Celtix Enterprise supplies two messaging brokers that support asynchronous messaging.

More information

For more information about developing asynchronous services see the section called "Asynchronous Invocation Model" in *Using Celtix Enterprise*.

Index

A

AMQP, 16
Apache Incubator CXF, 17

C

container, 18
 Celtix Enterprise, 18
 J2EE, 19
 JBI, 18
 servlet, 18

D

data bindings, 17
deployment
 containers (see container)

E

Eclipse SOA Tools Platform, 15
endpoint, 6
enterprise service bus, 6
ESB, 6

H

HTTP, 6
Hypertext Transfer Protocol, 6

I

intermediary, 16

J

J2EE, 19
JavaScript, 17, 23
JAX-RPC, 21
JAX-WS, 15, 17, 21
JBI, 18
JMS, 16
JMX, 15

R

routing, 19

S

service, 3
service engine, 16
service-oriented architecture, 1
servlet, 18
Simple Object Access Protocol, 6
smart endpoint, 23
SOA, 1
SOAP, 6
STP, 15

T

transports, 18

W

Web Service Definition Language, 5
WS-ReliableMessaging, 22
WSDL, 5

X

XML Schema, 5